(1)

DEPARTMENT OF THE AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

AFIT/GCS/ENG/90D-08

(1)

AN EMPIRICAL STUDY OF COMBINING COMMUNICATING PROCESSES IN A PARALLEL DISCRETE EVENT SIMULATION

THESIS

Ann Kathryn Lee
Captain, USAF

AFIT/GCS/ENG/90D-08

AFIT/GCS/ENG/90D-08

An Empirical Study of Combining Communicating Processes in a Parallel Discrete Event Simulation

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
In Partial Fulfillment of the
Requirements for the Degree of
Master of Science (Computer Systems)

Ann Kathryn Lee, B.S
Captain, USAF

December, 1990

Acknowledgments

I'd like to thank my advisor, Dr. Thomas Hartrum and my committee members Captains Robert Hammell and Catherine Lamanna for their advice, insight and encouragement this past year.

I'd like to thank Carol, Bob, Jason, and Brandon Fitch for making me a part of their family. I truly appreciate your encouragement and support and I'll never forget the 'quiet' family dinner discussions. I'm glad we're all a part of the same 'true' family.

I'd like to thank the entire Lee clan for all of their long distance encouragement. Keeping in touch with the family helped to keep things in perspective. I'd especially like to thank my grandmother, Annie Lucas, for her prayers and for her confidence in me.

Most importantly, I'd like to thank God for his steadfast love, his strength, and the peace he gives which passeth all understanding. Thank you, Lord, for seeing me through this experience and for all the life lessons you taught me that can't be quantified or measured.

Ann Kathryn Lee

Table of Contents

Page

List of Figures

List of Tables

AFIT/GCS/ENG/90D-08

Abstract

The primary goal of distributed discrete event simulations is to achieve speedup in simulation execution time by distributing the processing of the simulation over multiple processors. When partitioned for distribution in this fashion, simulations are typically partitioned such that there are more processes than processors. This thesis reviews existing methods for distributed discrete event simulations, and proposes general guidelines for efficient partitionings for a given communications topology based on empirical evidence.

A performance analysis is conducted for two approaches to partitioning the system. The first method chosen is a mapping of multiple processes to a processor and the second approach utilizes a distributed event list approach, developed by Mannix. This approach combines smaller processes into a larger single process, incorporating a next event list similar to that used in a sequential simulation.

Empirical studies compare the performance of the two approaches under a variety of conditions. The traditional Chandy-Misra approach to system partitioning is demonstrated to yield overall better performance than the distributed event list algorithm. General guidelines for partitioning the system for both approaches are developed based on the performance comparisons.

The impact of system topology on simulation performance is demonstrated. A connection between the system topology and the communication protocol implementation is investigated. The distributed event list algorithm is shown to exhibit a performance anomaly for a system network with directed cycles, and a correction for the algorithm is

proposed.

An Empirical Study of Combining Communicating Processes in a Parallel Discrete Event Simulation

I. *Introduction*

1.1 Why Distributed Simulation?

Computer simulations are used in various disciplines as learning, training, and analysis tools. Traditionally, simulations have been written sequentially and performed on sequential processors. However, as simulations have grown in size and complexity, there are those simulation models "whose computational requirements cannot be reasonably satisfied with even the fastest sequential processors" (24). Size and performance limitations of sequential processors are not the only motivation for the move to distributed simulations. As hardware technology has advanced, it has changed the dynamics of the cost/performance criteria which typically made up computer resource acquisition decisions, offering a broad range of tradeoffs in the way of cost versus performance considerations. The idea of harnessing the processing capability of several "affordable" computer processors as an alternative to purchasing a high cost high performance processor has been made feasible by advances in VLSI (very-large-scale-integration) technology, as well as advances in computer architectures. With the advent of parallel architectures and multiprocessor technology, there has been recent research into the use of parallel architectures for large simulations, which has led to efforts and interests in parallel, or as they are more commonly known, distributed simulations.

The basic goal of distributed simulation is to improve or optimize simulation performance, usually with respect to time and efficiency of operation. In many cases, simulation speed has been the factor of primary importance. Many of the applications for which this is true include weather modelling, fluid flow models and military simulations. These models, by the nature of the types of systems they represent, can be extremely complex and computationally intensive. When run on a sequential processor, these models often require exceptionally large blocks of 'real' time to simulate a few minutes or seconds of 'simulated' time. This increase in program execution time is unacceptable, because it limits the usefulness of the simulation in providing reasonable and timely feedback to the simulation analysts. For those applications where a reduction in size or complexity of the simulation results in a loss of simulation accuracy or realism, this is a problem. Distributed simulation provides a means to alleviate this problem.

Theoretically, if a sequential simulation is logically partitioned into separate processes, placed on separate processors and run in parallel, the amount of speedup attainable should be equal to the number of processors used. As is often the case with theory, actual implementation does not always yield the theoretical results. Research efforts seek to answer the questions (1) what are the obstacles to theoretical speedup? and (2) how can these obstacles be circumvented?

1.2 Simulation

1.2.1 Definition Simulation, as defined by Shannon, is "the process of designing a model for the purpose either of understanding the behavior of the system or of evaluating various strategies for the operation of the system" (27). Banks and Carson describe a simulation as the "imitation of the operation of a real-world process or system over time. Whether done by hand or on a computer, simulation involves the generation of an artificial history of a system, and the observation of that artificial history to draw inferences concerning the operating characteristics of the real system."(4)

1.2.2 Simulation Classifications The simulation models, according to Pritsker, have several classifications: discrete, continuous or combined. Discrete simulation occurs when the dependent variables of the simulation model "change discretely at specified points in simulated time". Discrete simulations can be further classified as event, activity and process oriented simulations, with the differences reflected by the way in which the objects or entities interact within the simulation model. Continuous simulation occurs when the dependent variables of the simulation model change continuously over time. Combined simulation occurs when dependent variables change either discretely, continuously, or a combination of both (22).

Neelamkavil further classifies simulations into two types, time-driven and event-driven, derived from the method by which the simulation time is advanced and maintained. In time-driven simulations, the clock is advanced from time t by a "uniform fixed time increment Δt ". In event-driven simulations, the clock is incremented "from time t to the next event time $\hat{t}$, whatever may be the value of $\hat{t}$" (19). The simulation type receiving the most attention in the literature with respect to ongoing research is discrete event simulation.

1.3 Issues of Concern

Many approaches have been described "for coordinating cooperating processes" within the simulation so that "the outcome of a parallel simulation is the same as would occur in a more conventional simulation (25) ." Distributed simulation assumes "a mapping from the physical processes being simulated to the logical processes that represent them in a simulation" and each of those logical processes must be optimally distributed to one of the processors of the parallel machine and executed (25). As stated earlier, ideally one would hope that the addition of n processors to the computational workload would yield a speedup of n, with each additional processor doing $\frac{1}{n}$ of the work required. However, parallel simulations rarely exhibit this kind of performance. This reduction in performance can be linked to three main causes: communication overhead, load imbalance and synchronization delay.

1.3.1 Communications Overhead The communications overhead incurred is the result of interprocessor communication necessary for logical processes on different processors to 'talk' to one another. "Communication overhead arises because intermachine communication is significantly slower then intramachine communication" (6). When compared to the sequential case, "any communication constitutes a penalty on the overall performance"(12). This is obviously due to the fact that a sequential process has no need to communicate with another process, and the realization that communication takes a processor's time that could otherwise be spent computing. If the cost of communication outweighs the benefit gained from partitioning a simulation or a similar process across several processors, no real speedup can be attained.

1.3.2 Load Imbalance Load imbalance is the result of improper allocation of logical processes to processors, such that the computational workload of one processor may be greater than that of any other processor. The speed of a parallel simulation will be bounded by the speed of the slowest processor; therefore, it is important to ensure each node has an equal amount of work to perform(12) . When load imbalance occurs, the uneven distribution of the workload results in a loss of both efficiency and projected gain in speedup; this loss is due primarily to processors with lighter workloads lying idle while there is still processing that needs to be done. Theoretically, those processors could divide the work remaining, resulting in an increase in both speed and efficiency.

Chu asserts that the load balancing and communication overhead are related issues, since processes must be allocated to processors not only to optimize the load for each processor, but also to minimize the overhead due to interprocessor communication. He outlines various strategies to balance these conflicting factors.(10)

1.3.3 Synchronization Delay Synchronization delay of local simulation clocks in discrete event simulations is another major concern in parallel simulation. Time-driven parallel simulations synchronize their clocks at every time step Δt. All events are therefore simulated in lock-step and no need for synchronization exists. Event-driven parallel simulations allow the clocks "to run out of synch, only synchronizing when necessary to prevent events from being processed out of order(14)." The primary synchronization algorithms under study follow either an optimistic or conservative strategy. Optimistic strategies allow the possibility of events arriving at a process out of order, and typically require a rollback strategy to correct improperly sequenced messages. Conservative methods allow processes to receive event messages in a monotonically increasing order, assuring no message will arrive from the 'past' to affect events within the process. This usually requires a process to wait on processing events to assure all event times are increasing.

While the problems of communications overhead, load imbalance and synchronization delay are not all inclusive as obstacles to optimal speedup, they represent a significant fraction of the research emphasis in parallel simulations.

1.4 Problem Statement

Chandy and Misra describe a paradigm to decompose a simulation to run in a parallel environment. This consists of partitioning the physical system and defining a discrete set of physical processes which communicate with one another exclusively via messages, and there is a logical process which corresponds to to every physical process. This mapping implies that every physical process is composed of the smallest and simplest discrete component possible in the physical system. Depending on the system being modelled, this implies the level of computation inherent to the physical process may be small in comparison to the level of communications for each logical process. Assuming that this is the case, one would suspect that any gain realized through parallelization could be lost due to excessive communication overhead. This has been documented in various implementations utilizing the Chandy-Misra null message strategy. (23)(24).

Mannix maintained that a system of N physical processes can be simulated by constructing a system of M logical processes, $M \leq N$, where each logical process simulates a disjoint set of one or more physical processes. He then utilized an event list, a classical data structure in event driven simulations, to maintain the proper sequential order of events within the simulation. Mannix demonstrated that this distributed event list, in conjunction with the null messages synchronization of logical processes, would yield significant speedup in some cases(17).

The goal of this thesis is to show using empirical evidence that partitioning a physical system is a matter of granularity, and the level of granularity, in conjunction with the communications topology of the system being modelled, plays a key role in simulation performance. In particular, this thesis explores the correlation between the communications topology of the physical system and simulation performance. The effect of different partitionings of the physical system into logical processes on both program implementation and simulation performance is examined.

1.5 Scope

This thesis effort will consist of empirical studies performed on discrete event simulations and conservative synchronization protocols. The Chandy-Misra null messages protocol(7) will be used. The effects of communications topology and its variations due to changes in granularity on simulation performance will be explored. All empirical studies will be performed on hardware implementing a distributed memory architecture.

1.6 Outline of Thesis

Chapter 1 will serve as an introduction and background. Chapter 2 contains a review of current literature. Chapter 3 will discuss the approach, methodology, algorithms, and cover design implementation of the experiments utilized in the research effort. Chapter

4 will be an analysis of experimental results and Chapter 5 will include conclusions and recommendations.

II. Issues in Distributed Simulation

2.1 Introduction

A discussion of parallel simulation requires an understanding of the major issues and terminology. This section outlines some of the issues involved in parallel programming in general, with emphasis placed on those issues central to parallel simulation.

2.1.1 Parallel Architectures Due to technological advances in general, and VLSI technology in particular, parallel computers exhibit architectures as varied as the types of applications that are available. Bertsekas and Tsitsiklis cite several parameters that can be used to describe or classify a parallel computer(5) :

- Type and number of processors
- Presence or absence of global control
- Synchronous vs. asynchronous operation
- Processor interconnection network
- Shared vs. local memory

The architectures of primary interest in parallel simulation are the SIMD (single instruction multiple data) and MIMD (multiple instruction multiple data) architectures. A SIMD machine is a processor structure which utilizes a single program to manipulate vectors or arrays of data(28). Applications which require repetitive operations on large amounts of similar data map well to this architecture.

MIMD machines are composed of several independent processors, each capable of executing its own program (28). Parallel simulations generally map most easily to this architecture, with each processor executing a portion of the simulation, communicating with other processors as necessary.

Applications, when decomposed for parallelization, can be mapped into a specific architecture based on the flow of program control, data decomposition, and the program algorithm. Chandy and Misra describe general heuristics and definitions for mapping programs to various architectures and schemas as early as the program design phase(9).

2.1.2 Synchronization In a distributed discrete event simulation, each process has its own local clock and local memory. The clock and local data variables are updated on a per process basis. It is necessary to coordinate the interaction and activities of different processes to ensure that the algorithm or simulation is executed in such a manner as to ensure the proper sequencing of actions. Synchronization consists of controlling the evolution of processes and therefore the occurrence of events(2). Synchronization is not an issue for a sequential application, because the order or sequence of events or activities is based on the order of program execution. In a system where each process is represented by a series of events, logical synchronization is defined to be the establishment of a form of agreement between processes which have arrived at a given event(2). Processes will be synchronized based on the synchronization protocol in use. These protocols generally fall into one of two categories: optimistic or conservative. Protocols from both categories will be discussed further in Section 2.3.

2.1.3 Interprocess Communication Interprocess communication in parallel programming arises from the necessity of exchanging control and data information between processes. The hardware method of communication will generally be determined by the architecture of the system being used. In parallel algorithms and systems, "interprocessor communication is a sizable fraction of the total time needed to solve a problem"(5). A communication delay or communication penalty is the result. Bertsekas and Tsitsiklis define the communication penalty as the ratio

$$CP = \frac{T_{total}}{T_{comp}}$$

where T_{total} is the time required to solve the problem and T_{comp} is the time attibuted just to computation given that all communications were instantaneous. The most important factors influencing communication delays are(5):

- The algorithms used to control the communication network
- The communication network topology
- The structure of the problem solved and the design of the algorithm to match this structure, including the degree of synchronization required by the algorithm.

2.1.4 Deadlock A system of processes is said to be deadlocked if there is at least one process waiting for an event that will not occur. When deadlock occurs in distributed simulations, the simulation cannot advance in simulated time order. Deadlock usually occurs as a result of message traffic between processes, where one or more processes will wait for a message not forthcoming from another process. A cycle of waiting processes results, and the simulation remains deadlocked unless some outside element intervenes. Parallel discrete event simulations depend on the synchronization protocol to resolve deadlock.

2.1.5 Load Balance Load balance is the idea of equally distributing the computational workload of the parallel program among processors for maximum throughput and efficiency. Any processors which become idle during program execution results in a loss of efficiency due to low processor utilization. Load balancing can be static, performed prior to program execution or dynamic, performed throughout program execution. Dynamic load balancing will cause additional overhead for program execution, because any time the processor spends performing the load balance is time that could be spent processing. For the simulation to remain efficient, the gain in balancing the load should outweigh any loss of processing time, with the net result of a faster, more efficient simulation.

2.2 Performance Goals

2.2.1 Speedup Speedup is the primary goal of distributed discrete event simulations. Stone defines speedup as "the ratio of time to execute an efficient serial program for a calculation to the time to execute a parallel program for the same calculation on N processors identical to the serial processor"(28). That definition maps into the following formula:

$$Speedup = \frac{Best\ serial\ time}{Parallel\ program\ time}$$

where the numerator is the running time of the most efficient serial program, and the denominator is the running time of the parallel program under study which maps to the serial program(28). One has to tread carefully when making claims of speedup to be certain that the benchmark being used is both accurate and understood so that useful conclusions can be drawn from the measurement. Different programs will exhibit different speedup curves, and this is generally a function of the inherent parallelism of the problem and how effectively that parallelism has been utilized in the parallel program.

The ideal or theoretical speedup thought attainable is n, where n is the number of processors used for the parallel computation. Speedup greater than n is defined as *superlinear speedup.* Superlinear speedup is rare, but has been documented in certain cases(17). Most speedup curves usually exhibit a degraded linear slope, with an initial linear slope that gradually falls as the number of processors increases. This is due to the fact that the compute time for a program can be divided into parallel and serial portions. No matter what the degree of parallelism that exists in a program, speedup will be asymptotically limited by that portion of the program which must be executed serially(1). As processors are added to work on the parallel portion of the program, the amount of work performed by each processor becomes increasingly insignificant.

2.2.2 Efficiency Efficiency is a program parameter. It is the measure of the proportion of time that processors are busy, and therefore it measures the impact of parallelization overhead on peak or ideal performance(28). Fox et. al. related efficiency to speedup by the following equation:

$$Efficiency = \frac{Speedup}{Number\ of\ processors}$$

and stated efficiency would be bounded above by one or more of four causes(12):

- Nonoptimal algorithm or algorithmic overhead - the parallel version of the algorithm in question may be inherently deficient.

- Software overhead - parallel implementations may require additional calculations unnecessary to the serial implementation.

- Load balancing - speedup is generally limited to the speed of the slowest processor, therefore ideally each node should perform the same amount of work.

- Communication overhead - as noted above, any communication in a distributed program constitutes a penalty in overall performance when compared to the sequential case.

2.2.3 Communication vs. Computation Stone asserts that "performance of parallel programs is strongly dependent on the ratio $\frac{R}{C}$, where R is the length of a run-time quantum and C is the length of communciation overhead produced by that quantum"(28). A low ratio implies poor performance due to a high communications overhead, while a high ratio implies the maximum level of potential parallelism may not be fully exploited. The ratio $\frac{R}{C}$ is then a measure of task granularity, with a low ratio corresponding to a fine grain parallelism and a high ratio corresponding to a coarse grain parallelism(28). For a fast, efficient simulation, a balance must be found in partitioning the simulation in a manner which fully exploits any inherent parallelism, while keeping the communication overhead to a minimum. This is the challenge of distributed simulation research.

2.3 Synchronization Algorithms

2.3.1 An Optimistic Algorithm - Time Warp Optimistic synchronization protocols allows processes to continuously execute events in the order in which they are received. If a process receives an event from the "past", the process will roll back its simulation time to the time the event should have occured. Once this has been accomplished, the process continues its processing from the adjusted simulation time. One such algorithm is the Time-Warp algorithm proposed by Jefferson(16).

The time warp algorithm is based on the paradigm of virtual time, "a global, one-dimensional, temporal coordinate system imposed on a distributed computation", which measures computational progress, defines synchronization and may or may not relate to real time(*16*). Processes *in the distributed system communicate with each other strictly* through message passing. The processes are effectively fully connected, possessing the ability to send messages to any other processor, including itself, without an established link or channel. Jefferson imposes two fundamental sematic rules on the system:

- *Rule 1.* The virtual send time of each message must be less than its virtual receive time.
- *Rule 2.* The virtual time of each event in a process must be less than the virtual time of the next event at that process.

Jefferson also places an implementation constraint on the system:

- *If an event A causes event B, then the execution of A and B must be scheduled in real time so that A is completed before B starts.*(16)

Each process has its own local virtual clock, which "changes only between events and then only to the value in the timestamp of the next message from the input queue"(16). Each process has an input queue, where incoming messages are stored in the order of their virtual receive time and the messages are processed, causing the process to execute continuously until a message is received from the "past". Once a message from the past is received, the process will "rollback" to the last stored time previous to the message which caused the rollback, and establish a process state suitable to that point in time. This will effectively undo any events local to the process. Antimessages are sent out for all event messages executed prior to receiving the message from the past. The antimessages provide the ripple effect necessary to bring each individual process back to a state where all processes can safely simulate again. The system state is kept logically intact by the concept of *global virtual time*, defined as follows:

- *Definition*: Global virtual time at real time r is the minimum of (1) all virtual times in all virtual clocks at time r, and (2) of the virtual send times of all messages that have been sent but have not yet been processed at time r.

Performance studies conducted using time warp(31) have demonstrated that significant speedup can be obtained for distributed simulations. The simulations under study have been combat simulations, which have exhibited irregularities in simulation algorithms, simulation time and object interactions. In spite of these irregularities, the model has performed well. Research continues in finding ways to optimize and improve performance.

2.3.2 A Conservative Algorithm - Chandy-Misra Conservative synchronization protocols wait until incoming events from other processes are received before they proceed with their program execution. The protocol assures the time order of incoming events so that no events can be received from the past. The waiting imposed by this paradigm may lead to deadlock in simulations. Research in conservative synchronization protocols tends to focus on the Chandy-Misra algorithm and its variants(8)(18).

The Chandy-Misra algorithm(8) is a distributed algorithm to facilitate program synchronization between processes for a distributed simulation. The algorithm assumes the physical system can be decomposed into physical processes (PP) which communicate with each other exclusively through messages. Each process in the physical system is modelled in the simulation by a logical process (LP), which is totally indepedent in its execution from the rest of the system(8). Operation of the LPs can be described in two phases: computing and communicating. When an LP is not computing, it is communicating or waiting to communicate.

Communication between logical processes is modelled after the time-stamped messages passed between physical processes in the physical system. Behavior of the messages received by a physical process is restricted by the following:

- a message arriving at time t cannot be affected by messages transmitted after time t
- a message sent to another PP at time t is dependent only on the internal processing of the physical process and all messages received up to time t

The paradigm restricts its attention to those physical systems displaying these properties, with the additional caveat that all physical processes possess an associated delay, $\varepsilon > 0$ between sequential message transmissions. This delay ε is the service or processing time associated with the physical process. All messages arriving at a process at time t are assumed not to produce an output from that process before time $t + \varepsilon$.

Deadlock in the logical system is resolved in one of two ways: deadlock avoidance using null messages or deadlock detection and recovery. In the first scheme, null or no-event messages are transmitted from LP_i to a communicating LP_j to inform LP_j that the logical time i has been updated. If LP_j has been waiting to receive a message from LP_i, the receipt of the null message will allow LP_j to update its logical clock and advance in simulated time. The deadlock detection/recovery algorithm allows the simulation to run in phases where it deadlocks, the deadlock is detected and a computation is initiated which allows LPs to advance their simulation clocks. The simulation continues until another deadlock phase occurs.

Contrasting the two approaches, the null message protocol enforces synchronization without the overhead of calculating deadlock and recovery, but with a significant communication penalty for additional null messages. The detection/recovery algorithm, although it avoids null messages, does so by diverting computational resources to the detection and recovery of deadlock. "Performance advantages depend on the relative costs of synchronization and message passing(24)." Reed conducted performance studies on a shared memory machine using simulated queueing systems. He reported that both algorithms exhibited poor performance for simulated queueing systems, and recommended it was not a viable approach for queueing applications(24). Fujimoto conducted performance studies using a distributed simulation testbed and a BBN butterfly shared memory machine. The simulation under study was a queueing network, and the parameters of the study included network topology and routing probabilities. Fujimoto reported positive results for both algorithms for most of the cases studied. He asserted that the amount of speedup attained depended heavily on the message populations and a calculated lookahead value, which is derived from the associated delay $\varepsilon > 0$(13).

2.3.3 A "Spectrum" of Synchronization Protocols Reynolds, after conducting several simulation experiments utilizing various synchronization protocols, asserted that there existed a "spectrum" of options for discrete event synchronization protocols, not merely the strict demarcation of optimistic and conservative protocols(21)(25). In addition, given this spectrum of options and a diversity of applications, there are an infinite variety of application and protocol pairings. He proposed to identify classes of protocols and classes of applications, and suggested that there existed bindings between classes that would lead to heuristics for choosing optimal protocols for a given application(21)(25). A simulation testbed, SPECTRUM(25), was developed to examine various simulations in combination with differing protocols in a common framework to make useful comparisons regarding simulation performance(21). The experiments also led to the development of a set of application(21) and protocol(25) characteristics and the suggestion that there is a dependence between applications and protocols that can be exploited for faster, more efficient simulation performance.

2.4 Recent Studies

2.4.1 Distributed Event List Algorithm Mannix developed an algorithm, the distributed event list algorithm(17), which utilized a classical event list structure within each logical process. The paradigm assumes a mapping of n physical processes to m processors,

$n > m$. Physical processes are mapped to logical processes such that each logical process contains one or more physical processes. The event list is used to enforce chronological ordering of events within each logical process. The Chandy-Misra null message protocol is used to enforce synchronization between logical processes.

Based on the performance analysis for the algorithm, "the implementation is shown to be of superlinear time complexity in relation to the events simulated. This implies theoretical speedup greater than N for a distributed simulation over N processors, contradicting the commonly held view of the existence of a bound of N on attainable speedup(17)".

Mannix performed empirical studies for the algorithm for queueing simulations under various conditions. He constructed a submodel (see Figure 2.1) which he connected in various topologies to form a complex queueing system. He demonstrated for certain topologies speedup greater than N was verified, and asserted that the topology of the simulation contributed directly to performance. Variations of the null messages protocol indicated that for tandem and feedforward topologies, a certain level of nulls was beneficial to simulation performance(11)(17).

2.4.2 A Hybrid Approach The idea of combining physical processes into larger logical processes was also proposed by Su(29). He tested the hypothesis by constructing a hybrid simulator (hybrid-1), consisting of several macroelements or multiple physical processes and utilized an event list to form a sequential simulator. The simulators were synchronized using a Chandy-Misra-Bryant variant protocol(29). The algorithm processes the event list in each sequential simulator subject to a "temporal marker" which indicates the smallest time for the arrival of an external event. The marker ensures no events are

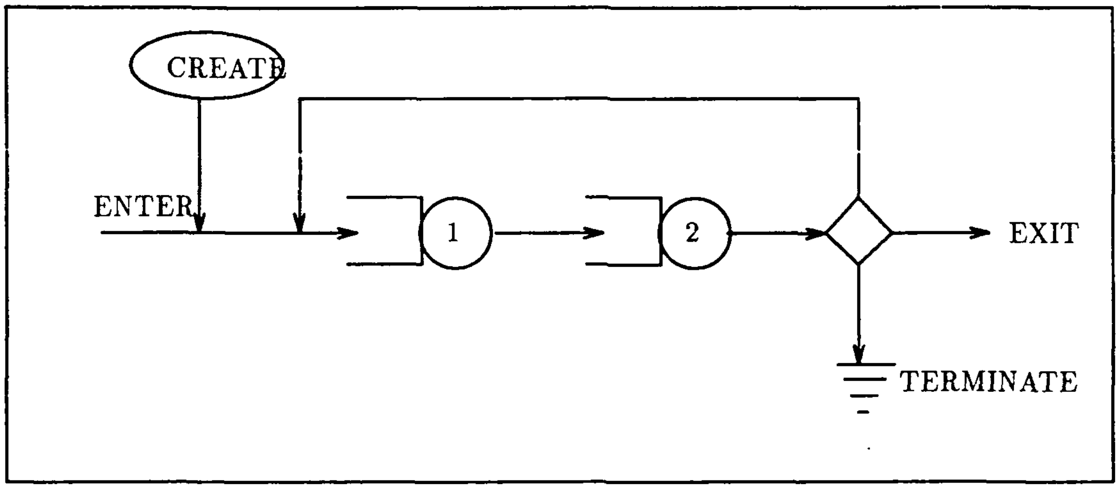

Figure 2.1. Mannix submodel

processed prior to a new arrival message from another macroelement(29).

Su performed an extensive study investigating the performance of logic simulations and variants of the Chandy-Misra-Bryant algorithms, which included a lazy message sending protocol, a demand driven operation with backward demand messages, as well as adaptive adjustment of lazy message sending parameters. The experiments were conducted in a programming environment Caltech developed for multicomputers, the Cosmic Environment and the Reactive Kernal(26)(29). Generally, simulation performance for a variety of logic networks utilizing the variants was very good, and a great deal of performance similarity was exhibited between the variants. The simulations exhibited near linear speedup in N initially, with an asymptotic drop as N increased and the available concurrency of the system was exhausted (30).

Contrasting the performance of the hybrid simulator to the previous results, Su asserted the hybrid simulator exhibited good speedup over all the variants in general. He also noted that if the elements were not properly distributed within a macroelement, the simulation time increased initially, before starting to decrease(30). It should be noted that the hybrid performance, while excellent overall, typically gave a comparatively poorer performance to the totally distributed algorithm.

2.4.3 Synchronizing on a Per Processor Basis Another approach on the same level of the hybrid paradigm and the distributed event list algorithm is that proposed by Bain(3). He proposed synchronization on a per processor basis, where n processes are distributed on m processors, $n > m$ and the synchronization was accomplished by using multiple synchronized event schedulers, one for each processing node of the system(3). All processes on the same node share the same event scheduler, consisting of a logical clock and a time-ordered queue of pending events. The event schedulers are in turn synchronized in a conservative manner, and no scheduler is allowed to advance until it is positive no process will receive an event from the past(3).

The event schedulers are fully connected and synchronized to the same global system time using multiple distributed spanning trees to collect and distribute clock times. The advantage of the full connectivity allows any process to communication with arbitrarily any other process, simplifying the partitioning of processes to processors. The algorithm exhibits a performance of O(D), where D is the diameter of system. On the Intel iPSC/2, the machine on which the experiments were conducted, this translates to

$$D = log_2 N$$

The one disadvantage of the algorithm is having to guard against race conditions that are possible in the implementation. The probability of encountering the race condition is a function of the communication patterns exhibited by the process interconnections, load balance, and system delays(3).

2.5 Summary

Speedup and efficiency are critical performance factors in evaluating a discrete event simulation. The potential speedup in a discrete event simulation will be determined by many factors: the level of inherent parallelism of the program, the synchronization protocol in use, interprocess communication, load balance, and the communications to computation ratio. Research efforts using conservative synchronization protocols have sought to increase simulation performance by increasing the level of computation in processes and decreasing the communication between processes for more efficient simulations. Effectively balancing the communications to computation ratio has resulted in a significant level of simulation performance speedup.

III. Approach and Methodology

3.1 Introduction

This chapter details the motivation and approach used for the empirical studies. It defines a logical process, discusses the algorithms and simulations implemented and gives an overview of the SPECTRUM environment.

When decomposing a physical system into a logical system of communicating processes, the objective is a resulting system that is well-balanced, having an efficient ratio of communications to computation resulting in increased performance. Finding that balance and simultaneously exploiting the inherent parallelism resident in the system is the objective of discrete event simulations, with the goal of obtaining a high level of concurrency for greater speedup.

3.2 Motivation for Combining Processes

In general, Chandy and Misra describe a paradigm where a physical system is "partitioned into physical processes (PPs) that communicate with one another exclusively via messages" and there is a "logical process (LP) corresponding to every PP"(7). This implies a "fine grain" partitioning of the physical system to a logical system of communicating logical processes. This is ideal for many applications using the Chandy-Misra protocol. For many discrete event simulations, it is often the case that there are more processes than processors available. Proicou(23) decomposed a VHDL application into numerous physical processes, partitioning his problem so that each VHDL process within the simulation was modelled as a separate logical process. Multiple logical processes were then mapped to a single processor node. In general, the simulations performed poorly, exhibiting 1:1 speedup with eight processors. Proicou asserted the poor performance of the simulations was the result of a lack of significant computation for each process, which was further inhibited by excessive null messages sent between processes residing on the same node. Later experiments with the simulation revealed that the poor performance was also attributable to excessive disk I/O. With that communication factor removed, the simulation exhibited a speedup factor of approximately 3:1.

Proicou suggested that a "coarse-grain" mapping of the physical system to a model where the LPs modelled multiple PPs would have improved simulation performance. He suggested that simulation performance would be enhanced by utilizing the distributed event list algorithm developed by Mannix for all processes which were mapped to a single processor node. Mannix maintained that a "system of N PPs can be simulated by constructing a system of M logical process with M $\leq$ N, where each LP simulates a disjoint set of one or more PPs". Once the physical system has been partitioned in this way, each LP "simulates events of its sub-model ... using an event list data structure and associated operations similar to those used in a sequential simulation"(17). Mannix asserted that this distributed event list, in conjunction with the null messages synchronization protocol, yielded speedups in some instances greater than N(17). Utilizing this paradigm in Proicou's experiment, this mapping would conceivably absorb the communications between processes on the same node, making intraprocessor communication unnecessary.

The distributed event list algorithm holds promise for those applications that need to map n processes to m processors, $n > m$, where the processes typically exhibit small computational loads. A portion of the communications overhead will be reduced due to the elimination of communication channels, especially between those processes residing on the same processing node. In addition, the aggregation of physical processes can potentially result in increased computation for the logical process, but this is dependent on the topology displayed by the combined logical process. This boost in computation combined with a drop in communications overhead theoretically results in increased performance. While the introduction of the event list to a logical process may potentially increase performance, a factor that must be considered is that it may also remove some the inherent parallellism achieved by the original partitioning of the system into simple processes. Given the initial results, this does not seem to pose a significant problem, provided a suitable partitioning of physical processes can be found.

3.3 Logical Processes

In the Chandy and Misra paradigm, each physical process in the system is mapped directly to a corresponding logical process in the simulation. This mapping implies that every PP is the smallest and simplest discrete component possible in the physical system. Depending on the system being modelled, this implies that the level of computation inherent in the resulting logical process may be very small in comparison to the communications. Assuming that this is the case, one would suspect that very little, if any, speedup would be realized within a parallel implementation due to communication overhead, particularly if the communications topology of the system dictates excessive communications between physical processes. If a more coarse grain partitioning of the physical system is to be realized in the context of the Chandy-Misra paradigm, it is necessary to redefine what was implied in the term *logical process*, and its relation to the physical system being modelled.

The smallest component within a physical system which can be logically modelled will be defined as a *physical process.* A physical process is the smallest viable computational unit which can potentially be mapped to a logical process. A *simple process* is the logical model of a physical process. A simple process maps directly to the Chandy-Misra definition of a logical process. A *logical process* can now be defined as a collection of one or more simple processes. The implication is that the LP consists of multiple smaller processes. The physical system to be modelled can now be decomposed into physical processes, mapped logically to simple processes, which are then modelled by logical processes in a discrete event simulation. This decomposition is summarized in Figure 3.1.

The following terms will be used throughout this document using the definitions given below:

- **Physical System** - the system or network being modelled
- **Physical Process** - the smallest component of the physical system that can be modelled as one of four general types of server processes: a router, a merge, a sink and a source (see Figure 3.2). Implicit to each simple process is the server queue, where the queueing discipline is application defined. In most cases, some processing or service delay is also implied.
- **Simple Process** - Logical representation of a physical process in the simulation.
- **Logical Process** - A collection of one or more simple processes within the logical system which performs an autonomous function. (see Figure 3.3).

The mapping of physical processes to a logical process is illustrated in Figure 3.3.

This "fine-grain" to "coarse-grain" partitioning of the system is significant in that it determines both the communications topology and the level of computation for the logical system. While it may not be immediately obvious, the topology of the logical system of processes has a significant impact on simulation performance. It is the hypothesis of this

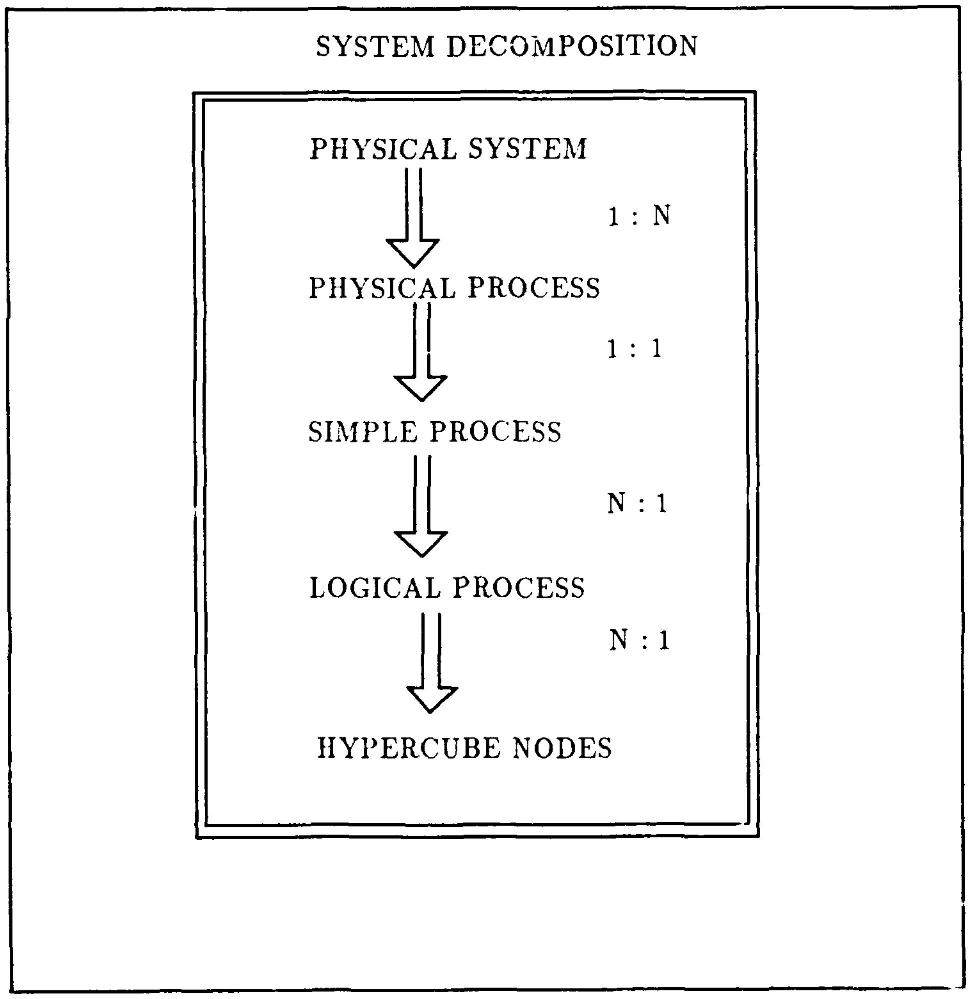

Figure 3.1. System Decomposition

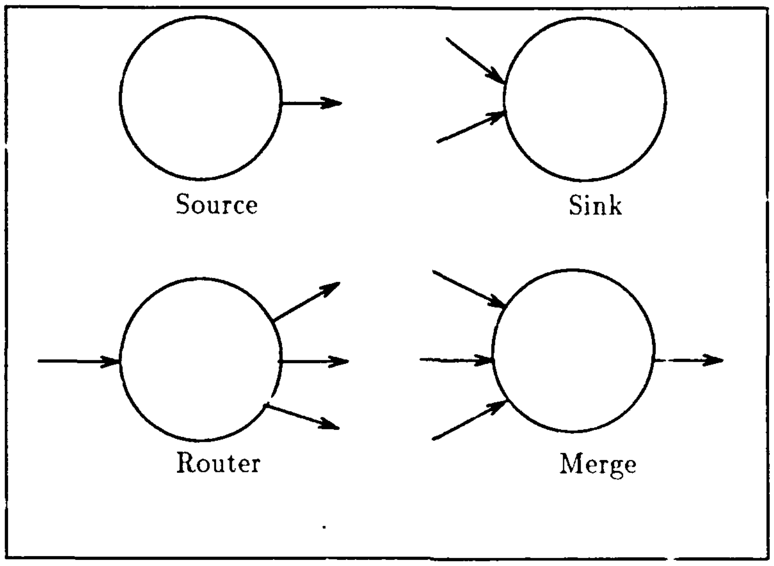

Figure 3.2. Four Simple Process Types

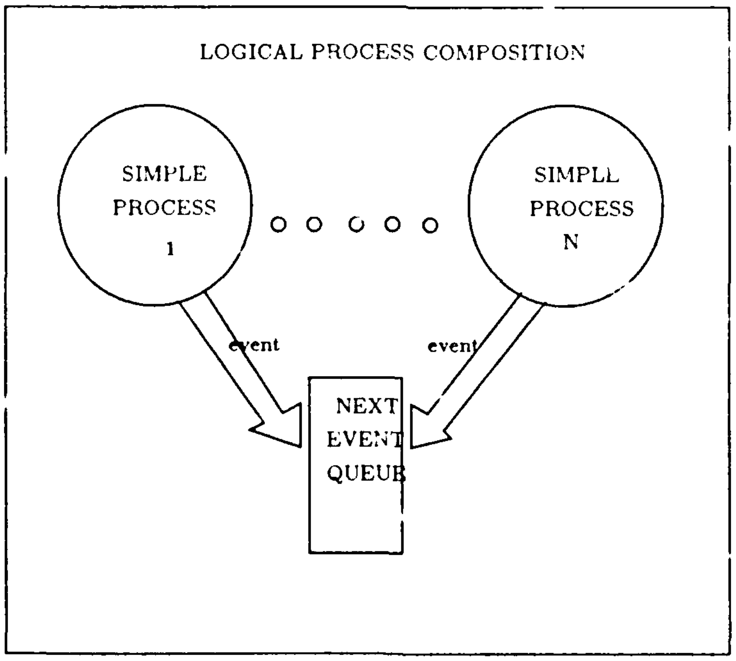

Figure 3.3. Logical Process Composition

thesis that the level of granularity in partitioning the simulation plays a key role in discrete event simulation performance.

3.4 Distributed Event List Algorithm

The distributed event list algorithm was designed by Mannix (17) with the idea that a system would be decomposed into n simple processes and mapped to m processor nodes, $n > m$. To reduce the communication overhead of interprocess communication between processes on the same node, simple processes can be combined with an event list and mapped to a single logical process, with i logical processes now mapped to m processor nodes, $i \geq m$, with $i = m$ as the ideal case.

Communication within the logical process so formed is accomplished as each simple process inserts event messages into the next event queue. Events are processed in time order as they reach the top of the list. No event is processed with a time greater than the simulation clock (*lp_clock*), including departing messages. A departure with time *t_out* > *lp_clock* is scheduled on the next event queue. The logical process advances its lp_clock to the time of the event popped off the list. This, coupled with the conservative synchronization between logical processes, ensures the proper progression of the simulation.

Correct simulation of the system is not guaranteed solely by the processing of the next event queue. An event may arrive at the logical process with an event time t which is less than the time of the next event on the queue. To ensure the correct ordering of events, within the logical process, it is necessary that LPs not process any event message on the next event queue until it is certain that no event will be received from a communicating LP which has a smaller time stamp. To accomplish this, *t_safe*, a lower bound estimate on the time of the next message arrival, is maintained. Events on the next event queue will be processed until the time of the next event exceeds *t_safe*. The value of *t_safe* is calculated by the protocol filter as described in Section 3.5.2.

The basic algorithm for the LP is as follows:

```
begin

    Initialize
        lp_clock, t_safe, t_neq = 0;
        Schedule any initial events

    while simulation not terminated loop

        Read Phase - Performs read until t_safe is updated
            while (t_safe ≤ lp_clock)
                Read and process incoming messages; update t_safe
            end while;

        Event Phase - Simulate all events up to t_safe
            while ((t_neq ≤ t_safe) AND (next_event_queue != EMPTY))
                next_event = pop(next_event_queue);
                lp_clock = next_event → time;
                Simulate next event
                    departure—consume event—enqueue new event
            end while;

        If no event is scheduled in interval (lp_clock,t_safe), advance lp_clock
            if (t_safe > lp_clock)
                advance lp_clock to t_safe;
            end if;

    end while

end
```

All synchronization between LPs is the responsibility of the synchronization protocol implemented. The details of the communications protocol are discussed in the Section 3.5.

3.5 Synchronization Protocols

The synchronization of logical processes in a discrete simulation is handled through synchronization protocols. Chapter 2 discussed the two more typical approaches to synchronization, conservative and optimistic. The protocol being used here is the Chandy-Misra deadlock avoidance strategy, which utilizes null messages between processes to properly synchronize processes and avoid deadlock. The Chandy-Misra algorithm assumes that all messages received by an LP are monotonically increasing, having a time greater than or equal to the current LP time. Logical processes which communicate with each other establish a communication line or channel, and each communication line has an associated clock value. For two processes LP_i and LP_j, the clock value i is a lower bound on the next outgoing message from LP_i and a clock value j is the last message received by LP_j over a communication line (i,j)(7). The synchronization protocol is responsible solely for the communications between processes. Theoretically, the protocol should be able to disregard the composition of the logical processes. However, in the actual implementation, the composition of the logical process may dictate some differences in the operation of the protocol. Two approaches were taken in the implementation of the protocol due to this phenomenon.

3.5.1 Chandy-Misra Filter-UVA version This implementation of the Chandy-Misra algorithm was supplied with the SPECTRUM testbed (see Section 3.6 for more information about SPECTRUM). The implementation is a simplification of the original Chandy-Misra algorithm. The notion of input and output channels and channel clocks is implicit, and not explicitly featured in the implementation. Another assumption made in the implementation is that each logical process is presumed to be a single server queue that experiences no waiting. The reading or waiting on input channels is as outlined in the following algorithm:

```
begin

    while (event message not received)
        Receive all pending messages.
            If a message has not been
            received from each input LP,
                block until message is received.
        Prioritize received messages by message time

        If message is a null message
            if event_time > lp_clock
                advance lp_clock to event_time
            if lp_clock has been advanced
                send a null message to all output LPs
                at time event_time + delay
            else if event_time + delay > lp_clock (New time information for LP)
                send a null message to all output LPs
                at time event_time + delay

        Else message is an event message
            return event message to LP

    end while

end
```

Event messages are sent or posted from within the logical process whenever a message is to be sent to another process. Null messages are sent

1. To all other communicating processes when an event message is sent out on one arc. The null message will carry the same time stamp as the event message.
2. When an LP receives a null message with new timing information (time = t), it sends out a null message at time t + delay.

The first condition ensures that when a message is sent to one logical process, the time of the event will be passed to all other logical processes. This information allows all communicating LPs to update their logical clocks without waiting, since the logical process will be unable to send another message at that time. The second condition performs the same function of informing a communicating LP that no messages will be forthcoming before the the stated time. The delay can be added to the original time of the null message because a delay of $\varepsilon > 0$ for each process is assumed, and any message arriving at the LP at time t is not expected to depart until time $t + \varepsilon$; therefore, the communicating logical process should not expect a message before that time.

3.5.2 The Event List Filter The addition of an event list to the logical process changed the basic assumptions presumed by the Chandy-Misra version developed by UVA, creating an incompatibility between the application and the filter. Instead of a single server queue with no waiting, the logical process consisted of one or more simple processes and an event list. In addition, the distributed event list algorithm (see Section 3.4) for the logical process incorporated the synchronization of the process as an integral part of the algorithm by not advancing the simulation clock past *t_safe*, which is calculated by the filter. Thus the event list filter has two primary functions: to calculate and update *t_safe*, preventing incorrect simulations; and to send null messages, preventing deadlock. This implementation conforms more closely to the original Chandy-Misra paradigm, explicitly implementing the concept of input and output channels for each logical process, and incorporating the notion of channel times.

The filter ensures correct simulation by updating the variable *t_safe* which is used to coordinate the processing of the next event queue and incoming messages. The filter updates the variable *t_safe* by reading the channel times of all input channels and taking the minimum value. This ensures that *t_safe* will have a minimum value such that no message will arrive from the past and no event on the next event queue will occur before an outside event arrives.

At any given point in the simulation, a logical process might read an unpredictable number of messages from a communicating LP. A constraint is placed on the number of messages a process is allowed to read to constrain the growth of the event list for that LP as well as limit the size of the input message buffer for that LP. The constraint is accomplished by allowing the LP to read and receive messages on those input channels with a channel time equal to the current *t_safe* value. The result is that the LP will check only those channels whose channel times established the lower bound for incoming messages during the previous read phase. An LP will read messages from input channels according to the following algorithm:

```
begin

    For all input channels
        while a message is pending and $next\_in = t\_safe$
            Read message on channel
            $next\_in = t$
            If the message is not a null message
                Insert message into event list
            end if
        end while
    end for

end
```

To ensure that deadlock is successfully avoided as proposed by the Chandy-Misra paradigm, Mannix outlines two null message conditions that must hold in the simulation:

1. For every logical process in the the logical system, once an LP exits a read phase of the distributed event list algorithm, that LP will send at least one message, either event or null over every output channel before entering another read phase.
2. If an LP sends an event message over a specified message channel (i,j), the LP must send a message, event or null, over every other output channel before sending another message over channel (i,j).

The first condition proposes to send null messages in conjunction with the receipt of messages, null or real. To meet the first condition, null messages are generated prior to updating *t_safe*. Null messages will be generated after each event phase in the distributed event list algorithm according to the following algorithm:

```
begin
    For all output channels which did not send
    a message during the last event phase
        Compute lower bound for next departure
            If the lower bound > last_out
                Send a null message on this output channel
                last_out = calculated lower bound
            end if
    end for
end
```

To meet the second condition, for any outbound message which is the result of an event departure on one channel, the filter will send a null message on all other output channels, to inform the communicating LPs of the latest departure and time. Although the current value of *lp_clock* provides a lower bound on the time of the next message to be transmitted over any outgoing channel of a logical process, the filter utilizes the channel times and a message prediction function to calculate a greater lower bound to allow the following LPs to advance their time as far as possible.

Calculation of this greater lower bound considers two possibilities. In the first case, an event on the next event queue may prompt a departure from the LP over an output channel. A lower bound for this possibility is the time of the next event on the event list, even though a departure event may be further down the list. In the second case, an event message departure may be prompted by some message which has not yet been received by the logical process. A lower bound on this possibility is the current time for the logical process added to the delay of the logical process or $lp\_clock + \varepsilon$. The delay for an aggregate logical process will be the minimum of all the simple process delays. Because each physical process in the system is assumed to have a delay $\varepsilon_i > 0$, each simple process will have a positive delay. Therefore, any minimum defined on this set of delays will be greater than 0. The calculated value $lp\_clock + \varepsilon$ is a minimum time for departures as a result of incoming messages because of the property of monotonicity, which implies the time stamp for all incoming messages is monotonically increasing. Given that all processes within the logical system exhibit a delay greater than 0, no message arriving at time t will depart prior to time $t + \varepsilon$. This implies that a lower bound on the next departure over any output channel will be calculated by

$$min(lp\_clock + delay, t\_neq)$$

Note that this requires that t_neq, or the time of the next event, be visible to the filter.

Describing a message channel (i,j) for communicating logical processes LP_i and LP_j, for each message transmitted over channel (i,j), LP_i calculates the lower bound for the next output message as outlined above and appends this time estimate to the message. All messages sent between LP_i and LP_j are of the form [t, m], where t is the lower bound estimate and m is the message, null or event, transmitted by LP_i. Note that t is distinct from the timestamp *t_msg* of the event message, which is included in m. For every output channel of LP_i, channel i is assigned the time t, designated *last_out* for any outgoing transmission. This same time t can be associated with the corresponding input channel j for LP_j when that logical process performs a read and is designated *next_in*. The value *t_safe* defined earlier for LP_j is $min(next\_in_{ij}) \forall i$.

3.6 The Simulation Environment

3.6.1 The Spectrum Testbed SPECTRUM is a testbed developed at the University of Virginia to study simulation algorithms in a common environment. SPECTRUM consists of three programming layers: the *process manager*, the *node manager*, and a *filter*.

The process manager provides a user interface to the application program. This includes functions for local clock advance, event management, and program initialization. The node manager is a set of functions responsible for machine level memory management and communications. The functions are machine dependent, providing SPECTRUM with portability to different machines without changing the simulation. The filter consists of a set of functions which implements the simulations communications protocol. These routines are called by the process manager and the node manager when communication is necessary between logical processes.

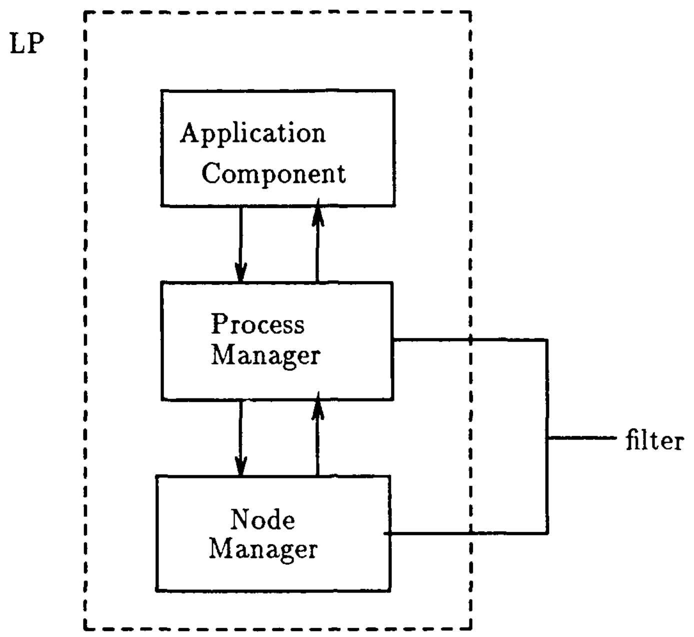

Figure 3.4. Composition of a SPECTRUM LP

The user supplies an application program which contains the simulation. The simulation is decomposed to application components, where each component corresponds to either a simple process or set of simple processes. The application component communicates with the process manager and filter through a combination of passed function parameters, function return values, and global variables. The application component, the process manager, and the node manger routines are combined to form a SPECTRUM logical process (see Figure 3.4) which is loaded on its respective node and executed. Logical processes are mapped to a SPECTRUM logical process as illustrated in Figure 3.5.

3.6.2 The iPSC/2 Hypercube The iPSC/2 Hypercube is a distributed memory architecture system. It consists of a front end processor and up to 128 nodes. Each node on the hypercube has the following features:

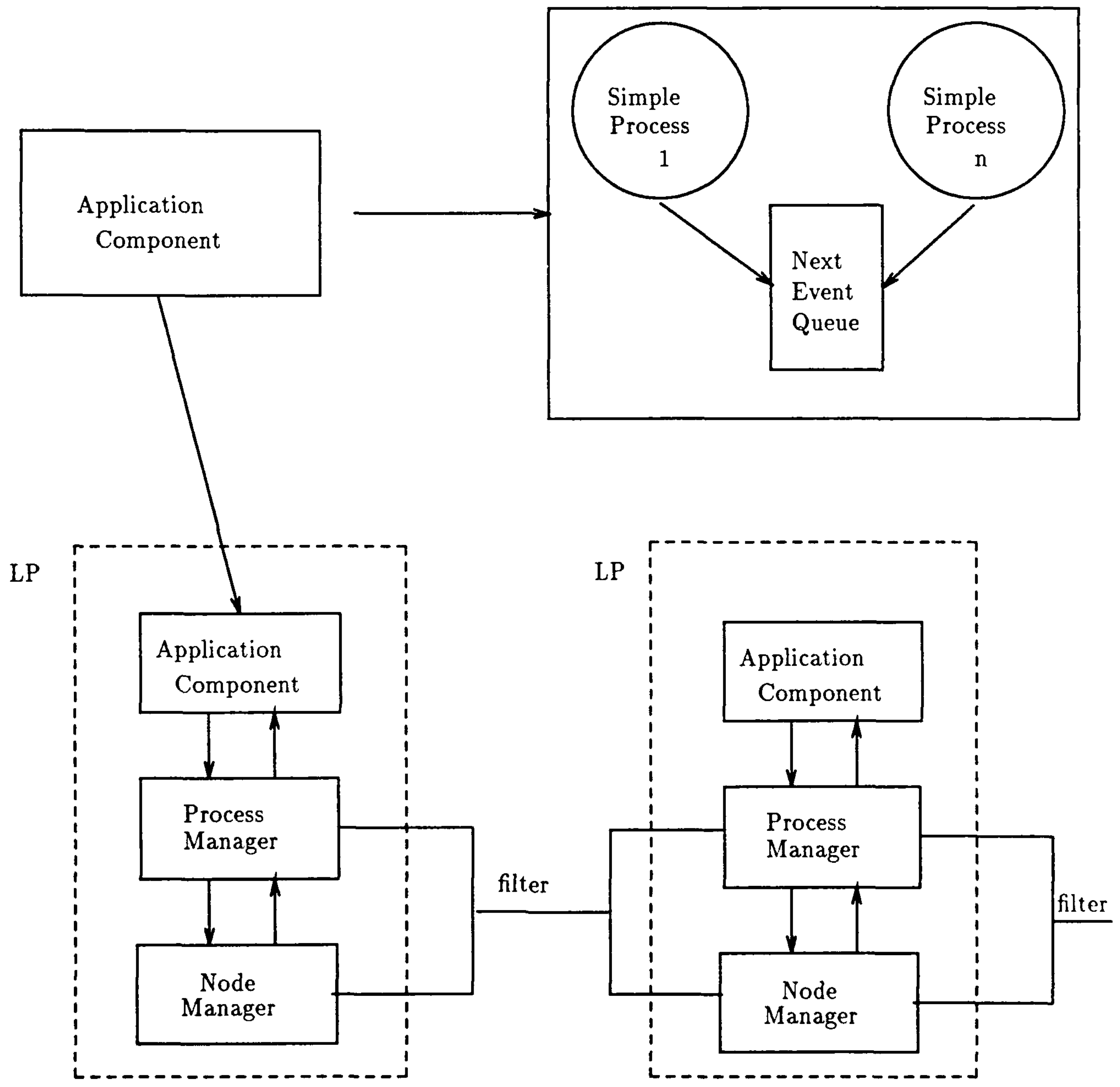

Figure 3.5. Mapping Logical Processes to a SPECTRUM LP

Message Size	Time (msec)	
(bytes)	process-to-process	node-to-node
0	0.332	0.353
65536	11.134	24.272

- 1, 4, 8, 12, or 16 megabytes of memory
- an 80387 numeric coprocessor (there are different options for high speed floating point arithmetic and vector processing
- a Direct Connect Module (DCM)

The DCM provides all of the nodes with direct message passing capability. "With DCM you can view the iPSC/2 system as an ensemble of fully connected nodes with uniform message latency(15)". Communication times between nodes are essentially uniform, even if the two nodes are two or more "hops" distant from each other. Through experimentation, it has been determined that the message timing between processes on the same node is virtually the same as the timing between processes on different nodes for small messages. For large messages (64K bytes) the time it takes to transmit a message between processes on different nodes is approximately double the time it takes to transmit a messages between processes on the same node (see Table 3.6.2).

The hypercube topology is designed so that nodes in the cube are connected by bi-directional links. A cube will contain 2^d nodes, where d gives the dimension of the cube, and each cube will be connected to d neighbors. Figure 3.6 illustrates a four dimensional (16 node) cube and its connections.

The AFIT iPSC/2 hypercube configuration is a three dimension (8 node) hypercube, with 12 megabytes of memory per node.

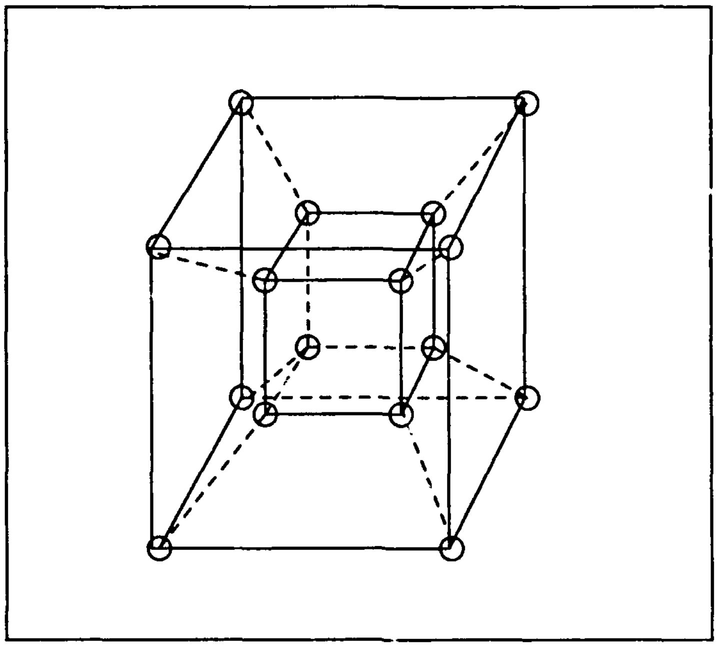

Figure 3.6. Four dimensional hypercube and connections

3.6.3 The Simulation The simulation used is a car wash, which is a simple queueing simulation composed of eight simple processes. There are three sources, four washes and a single sink process arranged in a feedforward topology, with feedback to two of the source processes (see Figure 3.7). Mapping each of the three process types to a process type defined earlier, the source process is a hybrid source-fork simple process. It "creates" cars and deterministically routes them to one of two wash processes. The wash process is a merge process. It simply adds a delay to simulate the wash time for the car and routes it to the exit. The exit process is a hybrid merge-router-sink process. When a car enters the exit, a determination is made whether to re-wash or consume the car. If the car is to be re-washed, it determines to which source the car will be sent. Otherwise, it consumes the car, with a print statement sent signalling the car has exited the wash. By the Chandy-Misra protocol, each process has its own clock and processes independently of the other

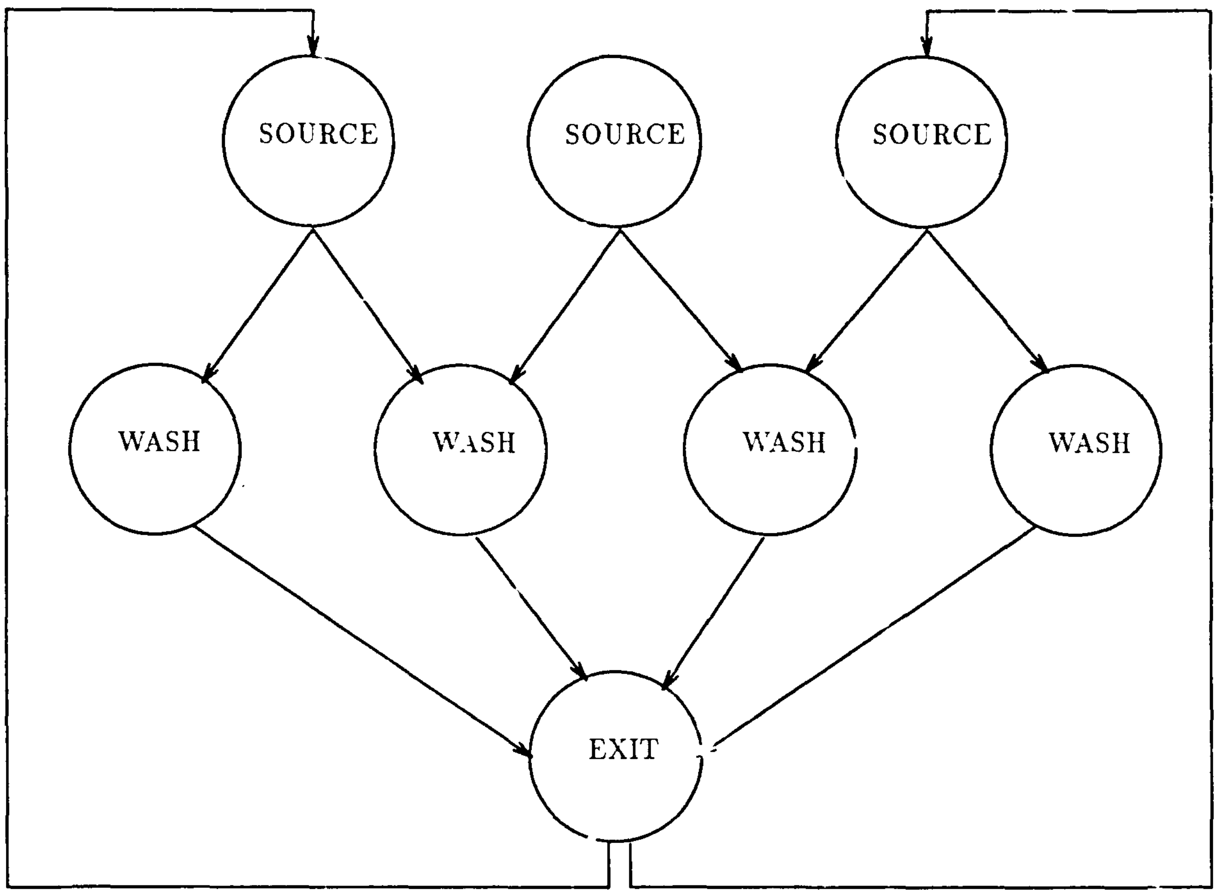

Figure 3.7. The Car Wash

processes in the system.

The car wash is a deterministic simulation, with all process delays set *a priori*. This was done so that each simulated run of the car wash would yield repeatable results and cars would exit in a predictable order. Once a testbench was established, each output from the implementation of the various topologies would be compared against the benchmark. In this way, a determination could be made of the "correctness" of the simulation. Any changes to simulation due to the topological configuration were detected by comparing the output files. Program errors were also detected using the output files.

3.7 *Summary*

The concept of a logical process has traditionally been mapped to a single process in the physical system, resulting in minimal computation for the logical process. Retaining that approach for the general decomposition of the system to be modelled, simple processes can be combined to form single logical processes with the objective of reducing communications within the system and possibly increasing the computation of the logical process. The effect of combining processes should result in a faster, more efficient simulation.

IV. Empirical Analysis

4.1 Introduction

This chapter details the experimental design, then describes and analyzes the performance. Rules of thumb for partitioning a phyical system are described. General observations and lessons learned from the experimental procedure are highlighted.

4.2 Experiment Design

The car wash was configured to run on 8, 4, 2 and 1 nodes. Two different methods were used. The first method maps multiple logical processes to each node. The communication paradigm assumed is the traditional Chandy-Misra approach. The filter used in this implementation was the Chandy-Misra filter supplied by UVA. In the 4, 2 and 1 node combinations, multiple processes were run on a single node.

The second method mapped a single logical process with a next event queue to a single node. The paradigm assumed is the distributed event list algorithm, in which each logical process is assumed to consist of multiple simple processes within the system which have been combined with a local next event queue. The filter used in this approach was a distributed event list filter developed for this thesis. In the 4, 2, and 1 node combinations, a single process was executed on each node. The eight node implementation for each filter utilized a round robin mapping of the processes, one per node. The topology of the car wash remained unchanged from the original.

Because the AFIT hypercube has eight nodes, the fine grain (one process per node) mapping of processes to node was exact for this simulation. Each process using the Chandy-

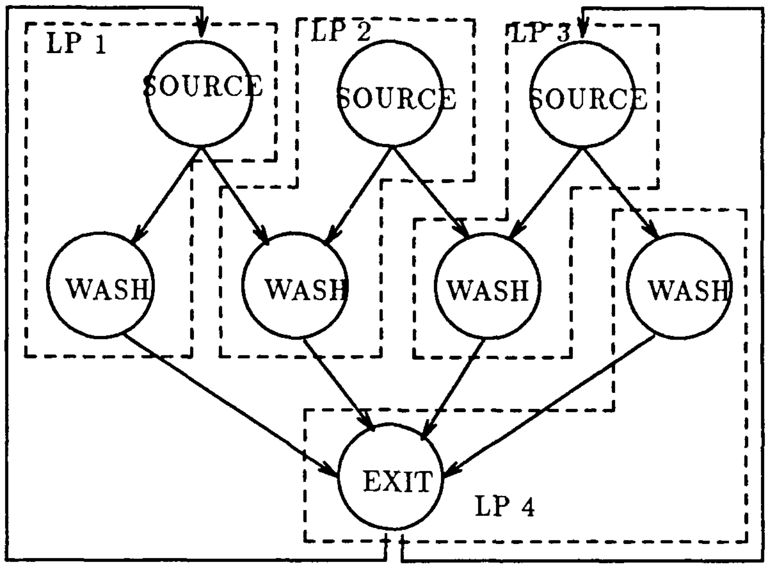

Figure 4.1. Four process configuration #1

Misra filter contained a simple process (i.e. a source, sink, or wash). Each process using the distributed event list filter contained a simple processs combined with a next event list. While there were many possible mappings of processes to nodes available, the 1, 2, 4, and 8 node mappings were selected because they evenly divide the cube, providing cube sizes of increasing dimension (0, 1, 2 and 3). Using combinations of two processes for load balancing, there were 28 combinations possible for the four node version. In the interest of time and scope, two configurations were chosen. The simple processes were grouped as in Figure 4.1 in the first configuration. This configuration was selected because it effectively consolidated communications lines between processes. Figure 4.2 illustrates the groupings for the second configuration. This configuration was chosen to exploit the inherent concurrency between groups of processes, attempting to examine the effect of combining processes that do not communicate and are systemically parallel.

Using 4 process combinations for load balancing, there were 70 possible configurations

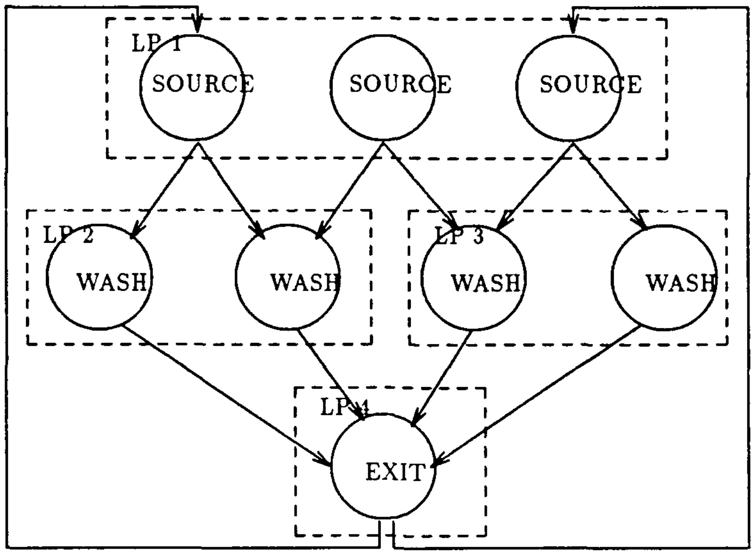

Figure 4.2. Four process configuration #2

for 4 process combinations for the two node version and two configurations were chosen. The simple processes were grouped as in Figure 4.3 for the first implementation. Again, this configuration was chosen to consolidate communications lines between processes. The second configuration (see Figure 4.4) was also chosen to examine the inherent concurrency of the topology.

The one node version for the Chandy-Misra filter consisted of placing all processes on a single node. The one node version used with the distributed event list filter served as the sequential baseline, with all processes consolidated into a single logical process with a next event list.

4.3 *Results and Analysis*

Several experiments were conducted. The focus and primary intent was to make a determination of how the simulation could be divided most effectively to promote speedup.

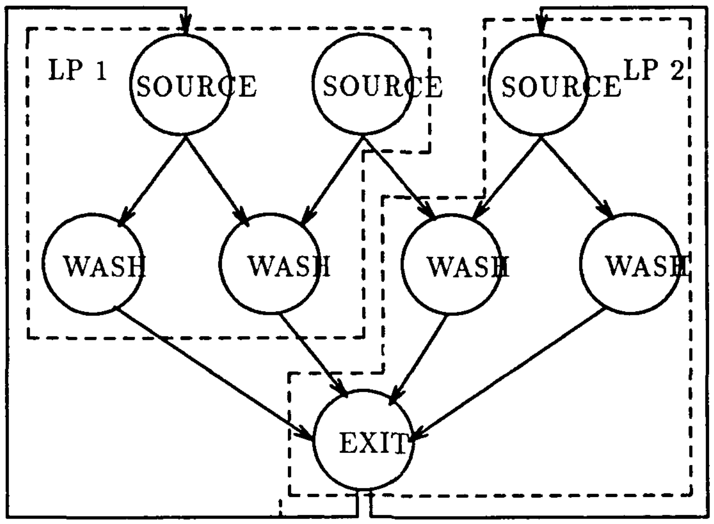

Figure 4.3. Two process configuration #1

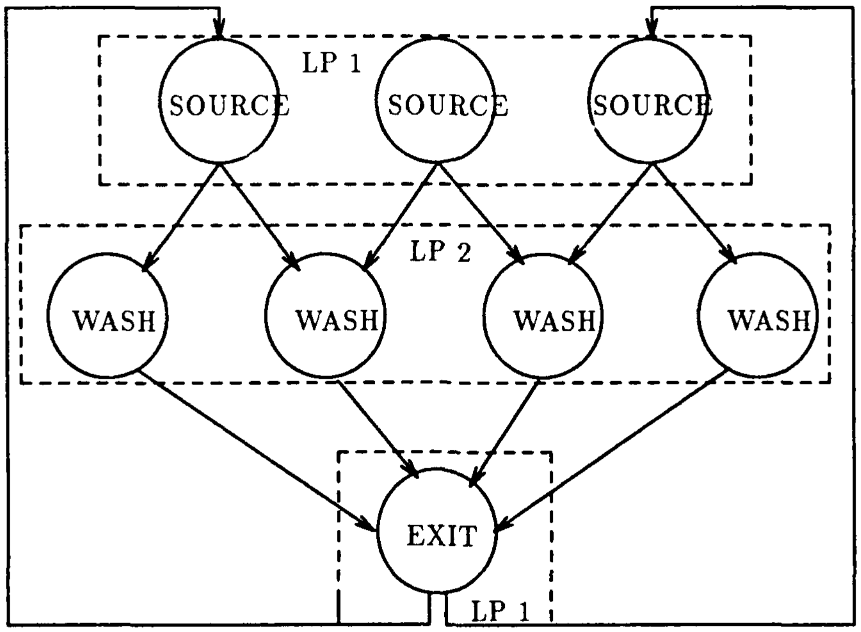

Figure 4.4. Two process configuration #2

A comparison of execution times for the various configurations was done in an attempt to answer the following questions:

1. If processes are to be grouped, what is the best allocation for these groupings?

2. How many nodes should be used to provide the most significant speedup?

3. Which paradigm exhibits the best performance? Under what circumstances?

After gathering an initial set of results, a general set of heuristics was developed, based on hypotheses formulated from the analysis of the data. To test the hypotheses, the same set of tests were run again on a slightly different topology (see Figure 4.8). The car wash was used again, this time without the feedback to the sources.

The results for the tests were analyzed using three different comparisons. The first comparison assessed the filter performance in each configuration. The second comparison assessed the impact of the configuration and process grouping on performance. Finally, each configuration was compared to the sequential version to assess the speedup attained. All comparisons were evaluated based on the best of three execution times of the simulation.

4.3.1 Comparison 1 This comparison assessed the performance of the Chandy-Misra filter versus the distributed event list filter implementation. This analysis focuses primarily on the filter, as well as noting some interesting performance differences for the permutations of the 4 and 2 node mappings. All four node mappings were compared. Since the 4 and 2 node versions had two configurations each, the execution times for the best (fastest execution times) configuration were used. All times are fastest execution time of

Table 4.1. Chandy-Misra Filter Run Times

Performance for Chandy-Misra Filter				
Logical Run Time	Average Execution Time (in seconds)			
	8 Nodes	4 Nodes†	2 Nodes‡	1 Node
1000	1	1	2	14
5000	24	6	24	na
10000	97	18	69	na
15000	299	32	97	na
20000	554	45	114	na

†Execution times are for configuration 2
‡Execution times are for configuration 2
na-no data available, run incomplete

Table 4.2. Distributed Event List Filter Run Times

Performance for Distributed Event List Filter				
Logical Run Time	Average Execution Time (in seconds)			
	8 Nodes	4 Nodes†	2 Nodes‡	1 Node
1000	1	2	1	1
5000	12	21	26	24
10000	41	71	98	95
15000	87	150	218	210
20000	151	261	384	372

†Execution times are for configuration 1
‡Execution times are for configuration 1

three trials. These times are displayed in Tables 4.1 and 4.2. The comparison for the 8, 4, and 2 node versions are graphically represented in Figure 4.5.

The distributed event list algorithm had the lowest execution times at the extremes, while the null messages algorithm exhibited the best performance at the intermediate levels. It was presumed that the execution times would be approximately equal across the board, with the execution times for the distributed event list algorithm being slightly better at the intermediate levels. However, the execution times for the Chandy-Misra filter are almost four times greater than the DEL filter for the eight node mapping (it failed to complete for run times greater than 5000 for the 1 node version) while the performance is much better than the DEL filter for the both the 4 and 2 node mappings. In the case of the 1 node mapping the disparity is easily explained by the fact that the Chandy-Misra implementation maps all eight processes to the same node and intranode communication for the single node is a bottleneck. The one node mapping for the distributed event list filter is the sequential model, which has no communication overhead.

The 8, 4 and 2 node mappings are less easily explained and not intuitively obvious. It is presumed that the number of messages generated throughout the simulation is most significant, especially in the case of the 8 node version. The number of messages generated in the distributed event list algorithm is almost three times greater than the number of messages generated in the null messages algorithm. This in itself was a perplexing phenomenon, but it explained the poor performance of the distributed event list algorithm. The algorithm has to cope with both the overhead for event list insertions as well as the tremendous communication load. Performance increases with the number of nodes due to the partitioning or shortening of the event list within the process. The disparity of the two intermediate mappings is linked most closely to the interprocess communications and the effect of the differing topologies, which will be discussed in the following section.

The hypothesis at this point is that, contrary to expectation, for a finer granularity or partitioning, the distributed event list algorithm yields good performance, but for overall performance improvement, multiple processes can be mapped to an intermediate number of nodes.

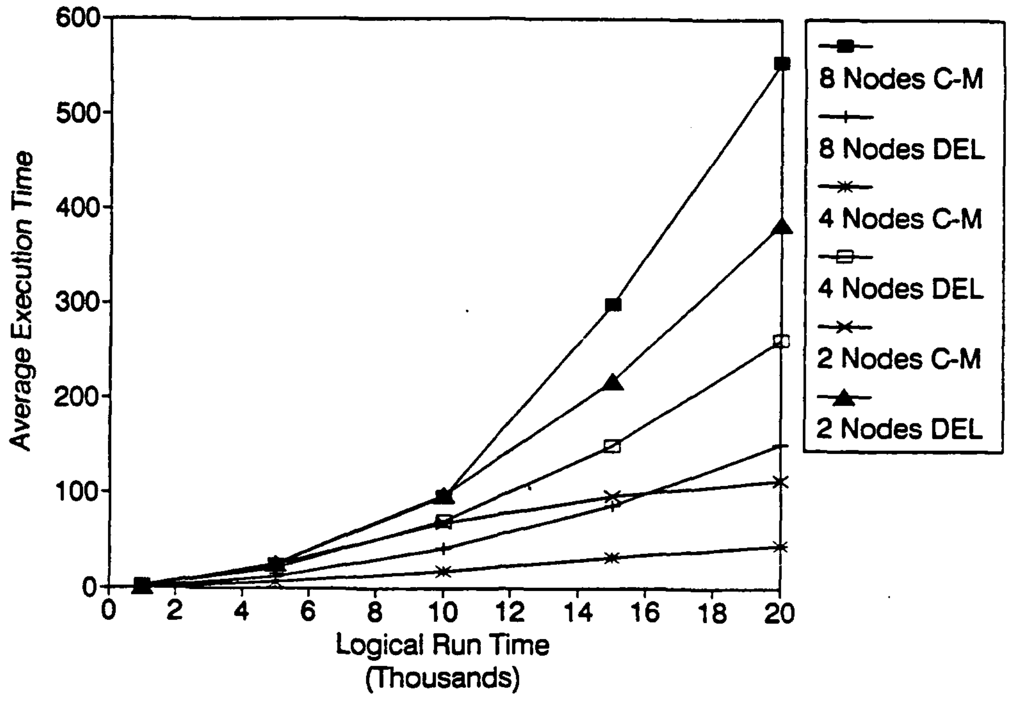

Figure 4.5. Filter Comparisons for 8, 4, and 2 node mappings

4.3.2 Comparison 2 This comparison assessed the performance of the different configurations on the 4 and 2 node versions of the simulation. The analysis focuses primarily on the effect of the topology of both the logical process itself (topology of the simple processes) and the resulting topology of the system once the processes were combined. The 4 and 2 node versions compared are the two configurations used for each. Results are found in Tables 4.3 and 4.4 and corresponding Figures 4.6 and 4.7.

Comparing both configurations for the 4 node mappings, configuration 2 outperformed configuration 1 using the Chandy-Misra filter. This performance is attributed to the fact that the first configuration groups processes that are directly connected by a communication line. Processes that are directly connected have an additional overhead of intranode communication which is not present in the second configuration. In addition, the

Table 4.3. Configuration Comparisons for the Chandy-Misra Filter

Logical Run Time	Average Execution Time (in seconds)			
	4 Nodes		2 Nodes	
	Configuration 1	Configuration 2	Configuration 1	Configuration2
1000	2	1	2	2
5000	37	7	na	24
10000	146	18	na	25
15000	324	29	na	66
20000	580	47	na	114

na-no data available, run incomplete

Table 4.4. Configuration Comparisons for the Distributed Event List Filter

Logical Run Time	Average Execution Time (in seconds)			
	4 Nodes		2 Nodes	
	Configuration 1	Configuration 2	Configuration 1	Configuration2
1000	2	2	1	3
5000	21	40	26	46
10000	71	155	98	177
15000	150	343	218	384
20000	261	611	383	710

second configuration takes advantage of the inherent parallelism of the simulation by partitioning so that those processes not dependent on each other for communicationto execute in a nearly parallel fashion, with no intranode communication to slow the processing.

The execution times of configuration 2 were more than double those for configuration 1 using the next event filter. The performance in this case was initially surprising, but less so on reflection. Since the distributed event list implementation combines the actions of simple processes into a single executing process, the first configuration effectively eliminates four communication lines and grants the logical processes more concurrency. In the second configuration, while the actual number of communication lines are actually fewer than the first configuration, the effective number of communication lines remain unchanged from the original topology with eight processes. Due to this phenomenon, the processes communicate more frequently. There are no communication bindings at all within the logical process and the logical processes are communication intensive, making communications the driving factor. There are filter implementation factors that also drive the performance of the simulation. The distributed event list filter spends quite a bit more time waiting when receiving messages from other processes. Input channels are read selectively to limit the size of the buffers required for input, as well as to ensure events are processed in the correct order. The Chandy-Misra implementation will read a single message from all input channels as long as a message has been received on each channel. As long as messages tend to arrive on all channels, the Chandy-Misra filter seems to spend less time waiting to communicate than the event list filter.

Comparing both configurations for the 2 node mappings gave the most interesting results. Configuration 1 using the null messages filter failed completely for any run time greater than 10000 logical time units. The distributed event list version for the same configuration exhibited the best performance, returning with the lowest execution times overall.

The first configuration for the distributed event list implementations returned the lower execution times of the two. This was surprising because the original presumption was that because the second mapping effectively removed the feedback loops within the system, as well as boosting the computation of the logical process, it would give the better performance results. Instead, it seems the second mapping failed to take advantage of the inherent concurrency within the system when the processes were combined into a single process. In addition, the logical process must now communicate more frequently, and it is the communication between nodes that once again dominates. This is the same phenomena exhibited in the four node mapping.

There were two general hypotheses formed from these comparisons. The first is that, when combining multiple nodes on a processor, it is wise to combine those processes which have no communications links on one node. Intranode communication in these instances are non-existent. The assumption here is that additional message traffic as a result of intranode communication overwhelms the buffers of the hypercube and intensifies the communications overhead. The general premise of the hypercube design is to optimize message-passing *between* nodes; it is assumed processes that communicate will be placed on different nodes. In a conversation with an Intel service representative, he stated that no one in his experience had placed communicating processes on a single node, so he had no real idea of the effect of intranode communication on message buffer behavior. He was of the opinion that stacking communicating processes on a node would be likely cause an

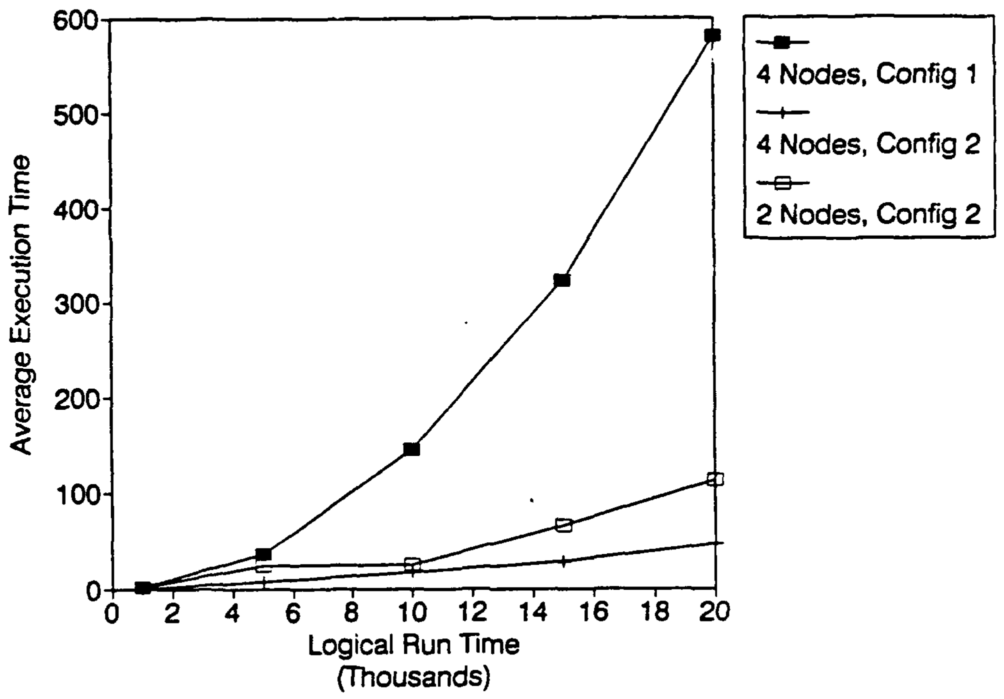

Figure 4.6. Configuration Comparisons for Chandy-Misra Filter

overflow for an excessive number of messages passed between the processes(20).

The second hypothesis pertains to combining processes effectively into a single process. In this scenario, it seems best to combine processes which share communication lines to the greatest extent possible. In addition, the more communication lines that are absorbed, the fewer nodes necessary. In addition to significantly reducing communication, this strategy boosts the logical process' 'computation'.

4.3.3 Comparison 3 The lowest execution times overall were displayed by the second configuration of the 4 node mapping using the null messages filter. This configuration exhibited a speedup of 3:1 or greater compared to the sequential version for run lengths with logical times of 5000 or greater. The next best mapping was the second configuration of the 2 node mapping using the null messages filter. The speedup factor in this instance

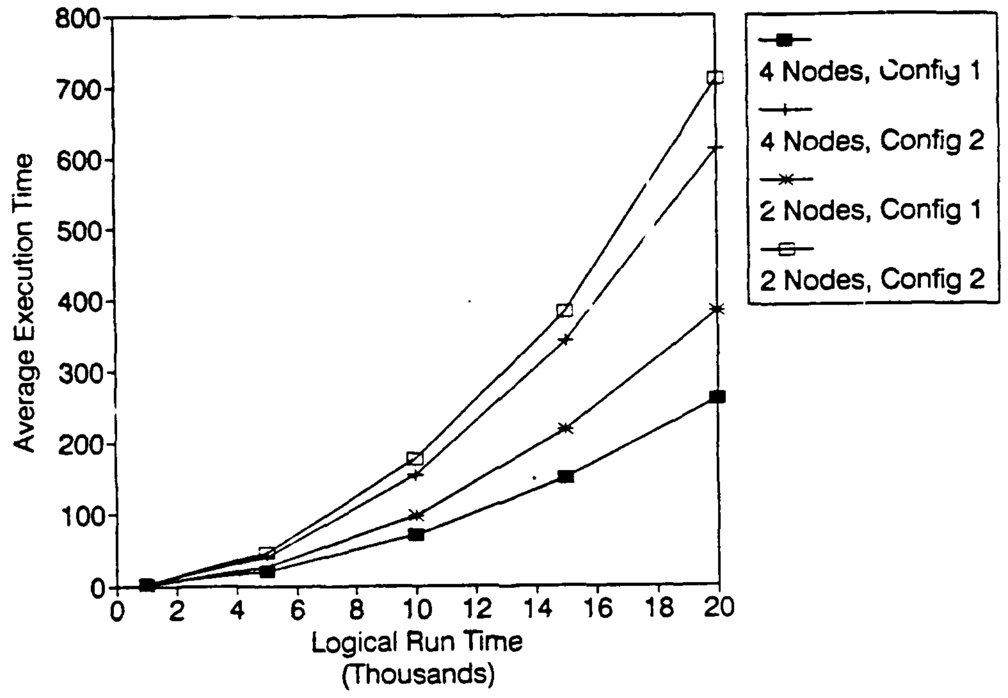

Figure 4.7. Configuration Comparisons for DEL Filter

was 2:1 for run lengths with logical times of 15000 or greater. The totally distributed version for the Chandy-Misra version was worse than times exhibited for the sequential version, while the distributed event list version exhibited a speed up factor of at least 2:1. As a general observation, the data seems to indicated that the sequential version of the simulation was very efficient. This may be credited to the lack of computation for the processes, as well as the feedforward topology of the simulation, which maps more naturally to a sequential flow of events. Under these circumstances, perhaps the expectation for a significant amount of speedup was overly optimistic.

A few general hypotheses can be made in looking at the execution times overall. First of all, using an event list within the logical process means that a sacrifice in performance is made due to a loss of concurrency in the partitioning. However, when distributed over an increasing number of nodes, there are significant performance gains. Overall, the best

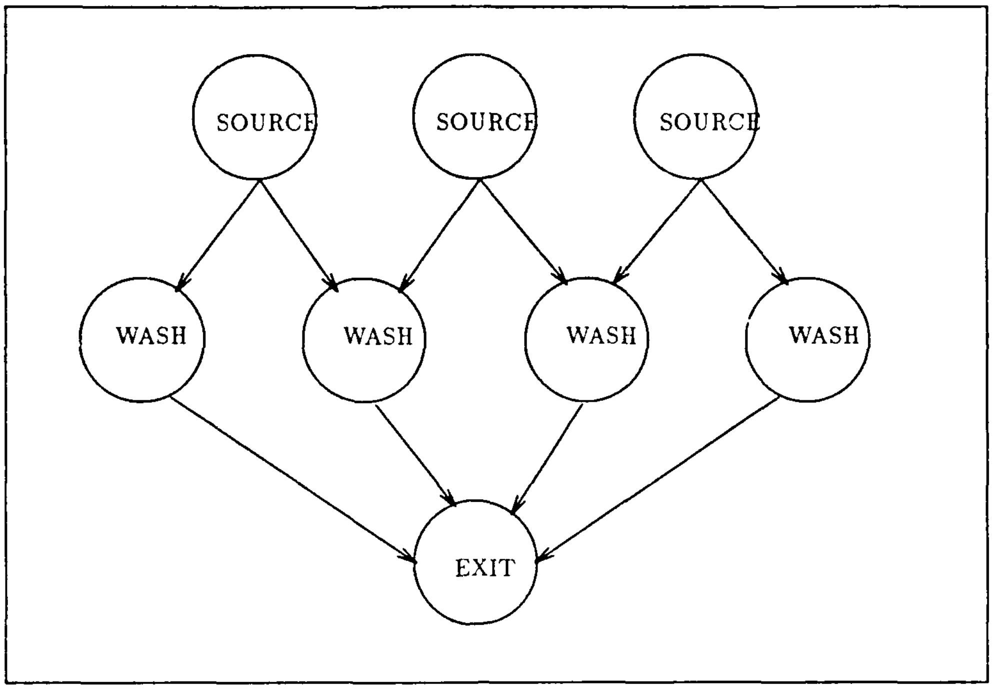

Figure 4.8. Topology 2-Car Wash without Feedback

performance is demonstrated when properly partitioned processes share a computing node.

4.4 Validation

Once general hypotheses were made concerning the performance of the differing configurations, the same experiments were applied again, this time to a topology without feedback (see Figure 4.8) to determine if the generalizations and data trends would hold for a different topology. The execution times for both filters are summarized in Table 4.5 and 4.6.

The expectation for this performance was that overall execution times would decrease, due to a reduction in the number of null messages generated as a result of removing the

Table 4.5. Chandy-Misra Filter Run Times-Without Feedback

Performance for Chandy-Misra Filter				
Logical Run	Average Execution Time (in seconds)			
Times	8 Nodes	4 Nodes	2 Nodes	1 Node
1000	3	3	2	26
5000	71	71	14	242
10000	356	322	43	611
15000	674	765	91	1126
20000	1212	1386	155	1829

†Execution times for configuration 2

‡Execution times for configuration 2

Table 4.6. Distributed Event List Filter Run Times-Without Feedback

Performance for Distributed Event List Filter				
Logical Run	Average Execution Time (in seconds)			
Time	8 Nodes	4 Nodes†	2 Nodes‡	1 Node
1000	27	1	1	1
5000	700	23	16	17
10000	3034	88	64	64
15000	6551	195	117	142
20000	11650	345	255	252

†Execution times for configuration 2

‡Execution times for configuration 1

feedback loops. As demonstrated by the data, this was not the case. Overall, execution times for a majority of the configurations increased significantly. This phenomena is best explained by the deficiencies of the simulation. The source processes of the implementation do not terminate after the maximum simulation time has been reached. Consequently, the sources run ahead of the rest of the processes and generate excessive messages within the simulation. With the removal of the feedback loops, the sources are never interrupted to process incoming messages and basically "run amok". The increase in the number of messages generated accounts for the overall increase in run times.

Several observations can be made comparing the two topologies. First of all, the performance for the second configuration of the 4 node mapping went from outstanding to poor, exhibiting a *slowdown* of more than 10:1 from the original topology. The performance of the second configuration of the 2 node mapping stayed fairly consistent, and demonstrated the best performance for this topology. In addition, both the 1 and 2 node mappings ran to completion for all points tested, unlike the feedback version of the same mapping.

Another observation to note is that the Chandy-Misra filter for the topology without feedback yields execution times that are twice that of the topology with feedback, in all cases except the 2 node version discussed earlier. In that case, the execution times are slightly less than or approximately equal across the topologies. The DEL filter also shows an increase in execution times (decrease in performance) overall for all the mappings. This behavior was not at all expected; on the contrary, it was expected that the removal of feedback in the system would provide speedup overall the configurations. Again, this increase in execution times may be linked to the increase in the number of messages generated. Of additional note, there was improvement in performance for those configurations that would not run to completion previously. Assuming that an overrun of message buffers caused the problems in the first configuration, this phenomena is doubling perplexing and at this time without reasonable explanation.

The next observation of interest is that the DEL implementation once again provides no real speedup over the sequential version. On the contrary, the 8 node version demonstrates a *slowdown* of almost 1:6 as opposed to the 1:3 slowdown of the feedback version.

The observation of most interest is that not all performance exhibited in the feedback version of the physical system is carried over in the system configuration with no feedback. In particular, both configurations of the 4 node mappings for both filters shift behavior radically. This leads to the hypothesis that the system topology of a simulation has a significant impact upon simulation performance and must be considered prior to partitioning a system.

4.5 Problems Encountered

As a general observation, more was learned from the implementation of the code before actually conducting the experiments. Problems with the implementation of various test configurations resulted in uncovering defects in the distributed event list algorithm, and revealed a relationship between the 'inner' topology of the logical process and the system topology, emphasizing the combined impact on simulation performance.

4.5.1 Simulation and Filter Connectivity Some of the initial problems with the car wash were due to a misunderstanding of the implementation of the filters. The primary purpose of the SPECTRUM testbed was to separate the application and the synchronization protocol. In this way, experiments could be conducted in a common environment, and any changes in simulation performance would be attributable to either the effect of changing the simulation or the effect of changing the synchronization protocol. After becoming more familiar with both the application, filter and interface routines, it became apparent that a strict separation of application from protocol is trivial in theory and very difficult in actual implementation. To properly incorporate the fundamental assumptions of the synchronization protocol, assumptions had to be made about the composition, homogeneity, and logical ope tion of the logical processes within the simulation. This was made obvious during the implementation of configuration 1 of the 4 node mapping utilizing the next event queue. The original intent of the experiment was to utilize a single filter, the original Chandy-Misra filter supplied with SPECTRUM, under the premise that the filter would be generic enough to support any LP configuration, especially when the application was unchanged. The original attempts at incorporating this filter for use with this LP configuration resulted in incorrect simulation results. It was initially conjectured that the combining of certain logical processes resulted in a fundamental change in the simulation itself, altering the simulation. Further debugging of the implementation revealed an inherent incompatibility between the newly defined logical process and the filter in use. This incompatibility led to the development of the event list filter.

The implication here is that the synchronization protocol and the application are intrinsically related. Furthermore, there exists a binding not only between applications and protocols, but possibly between the construction or topology of the logical process and the filter. While this may not apply to all communication protocols, there exists a very strong tie in the case of the Chandy-Misra algorithm.

After studying the Chandy-Misra algorithm at length, this binding may be explained by the fact that in this paradigm, the logical process is an active entity in the communication process. A logical process alternates between computing and waiting to communicate. A waiting process uses the following rules(7):

- The logical process waits on all input lines whose channel times equal the logical process' clock time.
- The logical process waits on all output lines where a message is ready to be sent.

SPECTRUM attempts to abstract the logical process to the point where the communications are transparent, but the logical process by implication has full knowledge and access to all input and output communication lines. While a logical process' computation may be abstracted from the remainder of the system, abstracting the communications between processes may only be possible to the degree of detail of how the communication is carried out at the machine level. The interface layer between the filter and the logical process must have access to information from both or the application and filter must share information. If an interface were used, the implementation would make use of information that is application specific. A generic interface at this point does not seem feasible. In both of the filters used in the experiments, application information was built into the filter. Another Chandy-Misra filter built to work with the VHDL simulation was also tried with this application. It worked well with the 8 node version, and returned execution times that were generally lower than those for the SPECTRUM Chandy-Misra filter. However, for the 4 and 2 node versions, the program generally deadlocked for all logical run times greater than 5000. The general speculation for the deadlock is that some assumption made in the filter holds for the VHDL application, but does not hold for the car wash application. Because a complete data set could not be gathered, this filter was abandoned for the comparisons and analysis.

Reynolds noted in his experiments with SPECTRUM that there is more of a binding between applications and protocols than was previously expected(21). It may not be possible to totally separate an application from its communications paradigm so that the two are totally independent. However, it may be possible to create a filter that is applicable to a class of applications, where certain assumptions about application properties are made of a set of applications in general.

4.5.2 The DEL Algorithm and Deadlock While implementing the 4 node configuration, it was found that the simulation would deadlock in configuration 1 and run to completion without any problems in configuration 2. It was thought at first that the system buffers were being saturated with null message traffic but testing ruled that possibility out. The system was deadlocked and this should not have been possible. Tracing through debug messages of the program uncovered a flaw in the distributed event list algorithm, which resulted in an anomaly for the simulation under certain circumstances and topologies.

Mannix states that two conditions are necessary to preclude deadlock in the simulation (see Section 3.5.2). The first null condition ensures that an LP sends null messages in conjunction with the receipt of messages from communicating LPs. The anomaly is the result of an inappropriate assumption concerning the calculation of the lower bound for the next outgoing message and the receipt of incoming messages by a logical process, which results in a violation of this null condition.

Based on the model in Section 3.5.2, two possibilities are considered during the calculation of the lower bound $min(lp\_clock + delay, t\_neq)$ for the next outgoing message. The first possibility is *lp_clock + delay*. This possibility will account for any departures from the logical process resulting from the arrival of a null or event message. It indicates that for any messages received by a logical process, a resulting departure from the LP cannot take place before *lp_clock*, which is a lower bound for the time of the arrival, and a positive *delay*, ε, which is assumed for every process.

The second possibility which must be considered is a scheduled departure on the next event queue. Since this departure requires no further processing, it will be sent when it reaches the head of the next event queue, at time *t_depart*. Since it hasn't been executed, $t\_depart > lp\_clock$, but it could be sent before the next arrival has been processed. Hence the *desired* lower bound is $min(lp\_clock + delay, t\_depart)$. However, the next scheduled departure, if it exists, could be anywhere in the next event queue, so that $t\_neq \leq t\_depart$. Thus *t_neq* is used in the calculation of the lower bound.

This lower bound gives correct results for all the configurations tested except the first four node configuration, where an anomaly occurs. The anomaly results from the assumption that *t_neq* is a consistently increasing value, which would guarantee that any sequence of calculated lower bounds would be monotonically increasing. Due to the fact that the queue is a priority queue, initially this does not appear to be an inconsistent assumption. What this model fails to take into account is the possibility of an arrival during the read phase which places an event at the *top* of the next event queue.

This opens the possibility that some calculation of $min(lp\_clock + delay, t\_neq)$ could result in a lower value than the previous calculation. Sending a message with this lower time would violate the assumption of monotonically increasing channel times. The anomaly is a direct result of the topology of the logical process. Using LP 4 as the example, the communications paths of the simple processes within the logical process defines multiple entry and exit points for the LP. Although these multiple entry points are effectively reduced to a single communications line between the two aggregate LPs, the anomaly is the result of the multiple destination processes available to an incoming message, which effects that event's eventual placement on the next event queue.

Correction of the anomaly needs to focus on the calculation of the lower bound for the next outgoing message. Because any departure resulting from a newly received message at time *t_msg* cannot occur until time $t\_msg + \epsilon$, a more accurate calculation of the lower bound is necessary. For any event on the next event queue, *t_neq* is not necessarily a departure. The information needed by the filter is not the time of the next event, *t_neq*, but the time of the next *departure*, designated *t_depart.* Therefore, the correct calculation for the lower bound of the time of the next arrival must be

$$min(lp\_clock + delay, t\_depart)$$

This calculation accounts for the next possible departure from the LP as a result of a newly received message, or a scheduled event departure from the queue. Now, any receipt of a message, null or event, should result in the appropriate calculation for the lower bound of the next departure, such that a null or event message will always be sent in between read phases for the logical process. The correctness of the proposed lower bound can be demonstrated to guarantee an increasing value for *t_depart*. The variable *t_depart* is determined from either a new arrival to the LP or as the result of an event currently scheduled on the next event queue. For a departure which is the result of a new arrival, *t_depart* is calculated $t\_depart = t\_msg + delay$. The delay, $\epsilon > 0$, is a positive constant and the property of monotonicity assures that any incoming messages for the logical process will be monotonically increasing, assuring *t_msg* is increasing. Therefore, an increasing value for *t_depart* is guaranteed. For a departure which is a result of an event currently scheduled on the next event queue, all events on the queue are increasingly ordered according to *t_msg*. For the case in which *t_msg* is an actual departure, *t_depart* increases due to the processing of the next event queue. For an intermediate event which may prompt a later departure, the increasing order of the events is guaranteed by the ordering of the next event queue. Any delay added to an intermediate event, resulting in the new value *t_depart* is positive; therefore, an increasing value for *t_depart* is guaranteed.

Deadlock in the system is caused by a two-fold effect, that of the logical process anomaly discussed above and the propagation of the effect of the anomaly throughout the logical system due to the system topology. In the current implementation, if the anomaly occurs, an out-of-order message won't actually be sent. Instead, the null message is *not* sent, depriving the following LP of the latest timing information. This in itself won't cause deadlock. The communicating logical processes may be bound together in a cyclic relationship. In the car wash scenario, because of the composition and topology of the

LPs in question, the communicating processes depend upon each other to advance their respective logical clocks. Any repetition of this scenario for any logical process in the cycle will result in deadlock for the entire system (see Figure 4.9).

This is a phenomenon not reported by Mannix, possibly due to the nature of the submodel he developed. The topology of the submodel itself is strictly tandem, with every simple process linked in tandem to the next simple process by a single link and an effective single connection with every other logical process within the system.

4.6 Summary

The results of the experiments in many cases were contradictory to performance expectations. The simulation deficiencies made it necessary to extrapolate information from the data trends and take those factors impacting performance into account. In spite of the problems, the experiments generated a few general guidelines for partitioning processes and validated the relationship between topology and its affect on communication and simulation performance.

Based on experience gained during implementation of the communications filters, a strong bond exists between the logical process topology and the communications protocol implementation. For the same application, a different filter was necessary to obtain correct simulation results due to differences in the topology of the logical process and the information required by the filter as a result of these differences.

In addition, it was noted from the experiments that the logical process topology plays as significant a role in the simulation performance as the overall topology of the system.

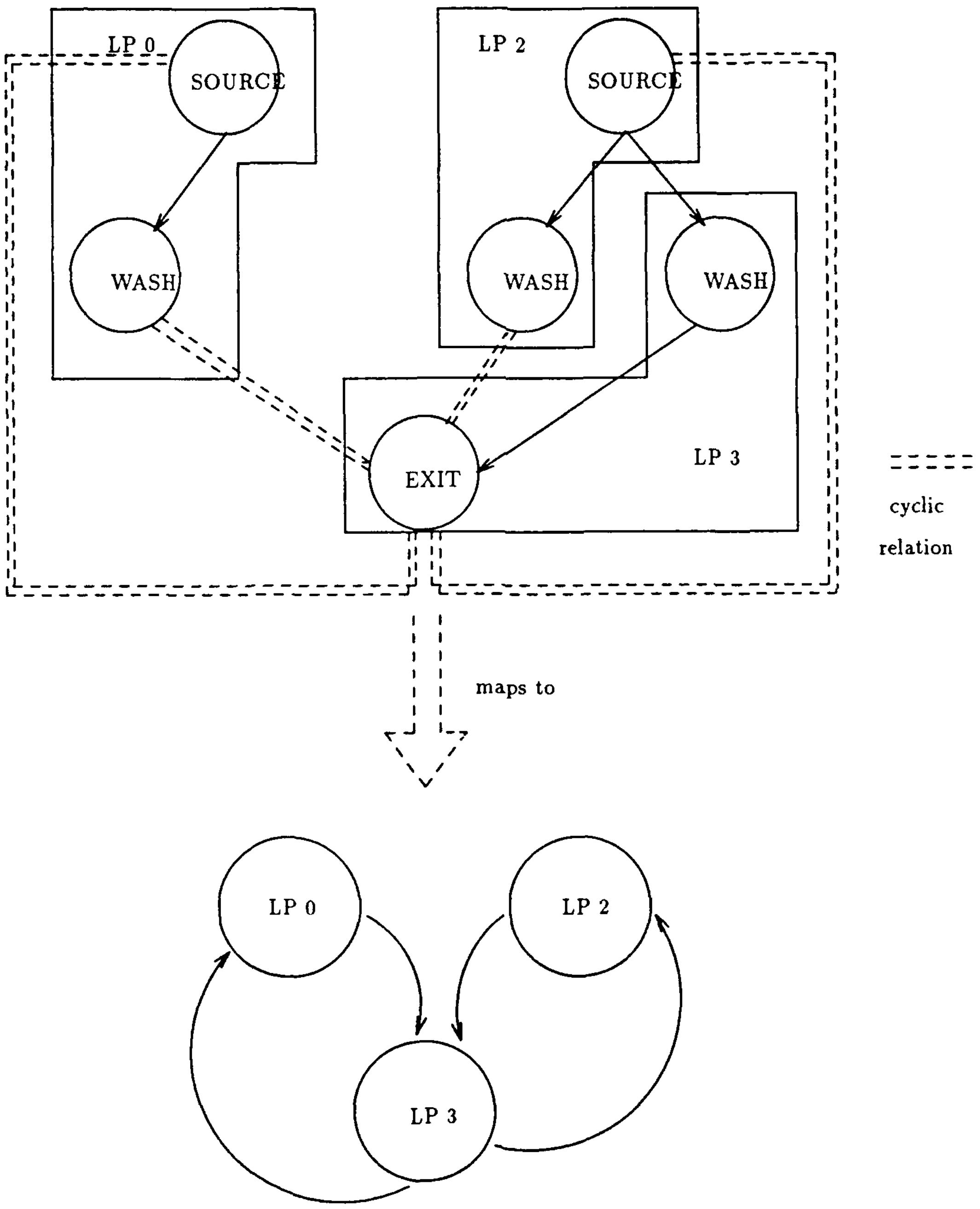

Figure 4.9. Relationship of Processes in Deadlocked System

This ‘inner’ topology affects the way logical processes consume and process event messages. The current distributed event list algorithm exhibits a performance anomaly resulting in deadlock for certain configurations. Although time did not permit revision of the code, the revision in the calculation of the lower bound for future message arrivals should correct this anomaly.

V. Conclusions

5.1 Overall Performance the Communication Algorithms

The performance of the null messages algorithm mapped to multiple processes was much better than originally anticipated. The general expectation was that regardless of most circumstances, because all of the processes were separate and distinct, there would always be so much communication between them that the communication factor would outweigh any mapping or partitioning considerations. Contrary to expectations, however, the strict null message implementation had the overall best performance for the scenarios investigated. What must be noted is that it is the overall impact of the topology and the partitioning of the workload, not necessarily the performance of the filter that plays the most critical role in contributing to the simulation execution time.

The distributed event list implementation did not perform as well as originally anticipated. An initial hypothesis for the poor performance is that it generates many more null messages than the null messages algorithm, (up to three times as many in this research). In addition, it was discovered that the topology of the communications within the logical process being modelled impacts the performance of the synchronization protocol. The algorithm maps several simple processes into an aggregate, larger logical process and in doing so implicitly imposes a new topology on the system being modelled, as well as a new set of event consumption rules for the logical process. The interaction of the t[illegible] made it necessary to customize the synchronization protocol to account for the next event queueing and the aggregate processes. A performance anomaly of the filter under certain topologies uncovered a flaw in the distributed event list algorithm which can potentially cause deadlock in the system when configured for certain topologies. The algorithm gave correct results for all configurations which did not deadlock. For the configuration that did deadlock, it was possible to fine tune the application and logical process delays to avoid the anomaly by using a more conservative time lower bound to give correct results for the simulation.

5.2 Topological Effects

When partitioning a system so that multiple processes will run on a single node, the general rule is to take advantage of processes that have no communications lines common and in addition can be run in parallel. Lack of communication lines implies that the only communication will be communication between processes on other nodes. Since processor cycles are already split between the processes sharing the processor node, no cycles are wasted on communications between processes on the same node, and processes need not wait on each other.

The converse is true if the desire is to aggregate the processes into a single logical process. The lack of communications in this case destroys the inherent parallelism within the system and increases the message traffic between logical processes. The approach to utilize in this instance is to combine those processes which share communication lines. This increases computation and decreases communications, as long as the aggregate processes are loosely coupled based on the system topology. If the logical processes are communication dependent, there is less concurrency in the resulting system and performance tends to drop. An important result here is that the choice of multiple processes vs. a single process per node results in *different* partitioning schemes. The system designer is cautioned against tuning the system performance using multiple processes (which is easier to do), and then combining those processes to improve simulation performance.

Feedback loops within a topology may not always create additional null messages within a system. Simulation implementation deficiencies skewed the results for the no feedback configuration, leading to less than conclusive results from which a meaningful hypothesis can be made. However, the data does point out the fact that for processes with little computation, the *absence* of feedback which serves as the sole source of input for a process may allow that process uninterrupted, freewheeling computation. If this process is responsible for generating messages within the system, this freewheeling computation may turn into freewheeling communication and cause an explosion of messages within the system. The impact of the feedback loop in the system will depend not only on the overall topology of the logical system but the composition or function of the logical processes within the system as well.

5.3 Recommendations for future research

The SPECTRUM testbed is a good tool for conducting experiments in a common environment, but many improvements could be made to the overall environment to make it more useful. The SPECTRUM application code and the Chandy-Misra filter implementation should be modified for realism and to more accurately match the paradigm. Incorporating data collection of message passing statistics and message buffering information into SPECTRUM would be useful for data gathering and program debugging. In addition, the current implementation of the distributed event list filter should be corrected as outlined in Chapter 4.

Further investigation should be made of the level of dependence between the application and the filter. In this research, the application required information from the filter for proper ordering of events and the filter required information from the application to assure proper synchronization. Is it possible to design a filter to fit any general application? What is the minimum interface necessary between an application and the filter?

This research uncovered several observations that should be explored. The performance of the distributed event list algorithm was extremely poor, particularly for the topology with no feedback. Is this behaviour related to implementation defects or is there some other explanation? There were some configurations which failed to run to completion in the feedback topology, yet ran without problem in the no feedback topology. Assuming that saturation of message buffers caused problems in the first case, if more messages were generated in the second topology, why did the problem disappear? These inconsistencies merit further investigation.

There is much more to be explored. Time constraints did not allow an evaluation of the effect of spin loops on performance. This is yet another factor of performance that must be considered. The experiments for this study focused on a single application with an evenly balanced topology. Future research should include a wider range of applications and experimentation with topologies that are more challenging than the topology studied here. The number of processor nodes available here is a limiting factor and future experiments should investigate with a larger number of processing nodes. Future investigations should include analyses of more complicated, computationally intensive simulations to discover if the stated generalities hold over a greater set of applications.

Bibliography

1. Almasi, George S. and Allan Gottlieb. *Highly Parallel Computing*. The Benjamin/Cummings Publishing Co., Inc., 1989.
2. Andre, F., et al. *Synchronization of Parallel Programs*. MIT Press, 1986.
3. Bain, William L. "Parallel Discrete Event Simulation Using Synchronized Event Schedulers." In *Proceedings of the Fifth on Distributed Memory Concurrent Computers*, pages 90–94, April 1990.
4. Banks, Jerry and John S. Carson. *Discrete Event System Simulation*. Prentice Hall Inc., Englewood Cliffs, NJ, 1984.
5. Bertsekas, Dimitri P. and John N. Tsitsiklis. *Parallel and Distributed Computation: Numerical Methods*. Prentice Hall, Inc., 1989.
6. Bokhari, Shahid. *Assignment Problems in Parallel and Distributed Computing*. Kluwer Academic Publishers, 1987.
7. Chandy, K. M. and Jayadev Misra. "Asynchronous Distributed Simulation Via a Sequence of Parallel Computations," *Communications of the ACM*, *24*(11) (April 1981).
8. Chandy, K. Mani and Jayadev Misra. "Distributed Simulation: A Case Study in Design and Verification of Distributed Programs," *IEEE Transactions on Sofware Engineering*, *SE-5*(5):440–452 (1979).
9. Chandy, K. Mani and Jayadev Misra. *Parallel Program Design: A Foundation*. Addison-Wesley Publishing Co., Inc., 1988.
10. Chu, Wesley W. and others. "Task Allocation in Distributed Data Processing," *Computer*, *13*(11):57–69 (1980).
11. Davis, N.J., et al. "Distributed DIscrete-Event Simulation Using a Null Message Algorithm on Hypercube Architectures," *Journal of Parallel and Distributed Computing*, *8*:349–357 (1990).
12. Fox, G., et al. *Solving Problems on Concurrent Processors*. Prentice-Hall, Englewood Cliffs, New Jersey, 1988.
13. Fujimoto, Richard M. "Performance Measurements of Distributed Simulation Strategies," *Distributed Simulation 1988*, pages 14–19 (1988).
14. Hartrum, Thomas C. "Parallel Simulation Research at AFIT." AFIT Parallel Simulation Group Summary, December 1989.
15. Intel Corporation. *iPSC/2 User's Guide*, March 1989.
16. Jefferson, David. "Virtual Time," *ACM Transactions on Programming Languages and Systems*, 7(3):404–425 (July 1985).

17. Mannix, David L. *Distributed Discrete Event Simulation Using Variants of the Chandy-Misra Algorithm on the Intel Hypercube.* Technical Report AFIT/GCS/ENG/88D-14, Air Force Institute of Technology, 1988. NTIC # AD A202 849.

18. Misra, Jayadev. "Distributed Discrete Event Simulation," *ACM Computing Surveys*, *18*(1):39–65 (March 1986).

19. Neelamkavil, Francis. *Computer Simulation and Modelling.* John Wiley and Sons, 1987.

20. Nessar, Steve, "Intel Service Representative." Telephone conversation, 20 November 1990.

21. Paul F. Reynolds, Jr. and others. "Comparative Analyses of Parallel Simulation Protocols." In *Proceedings of the 1989 Winter Simulation Conference*, pages 671–679, December 1989.

22. Pritsker, A. Alan B. *Introduction to Simulation and SLAM.* John Wiley and Sons, 1986.

23. Proicou, Michael C. *A Distributed Kernal for Simulation of the VHSIC Hardware Description Language.* Technical Report AFIT/GCS/ENG/89D-14, Air Force Institute of Technology, 1989. NTIC # AD-A215 419.

24. Reed, Daniel A. and Allen D. Malony. "Parallel Discrete Event Simulation: The Chandy-Misra Approach," *Distributed Simulation 1988*, pages 8–13 (1988).

25. Reynolds, Paul F. "A Spectrum of Options for Parallel Simulation." In *Proceedings of the 1988 Winter Simulation Conference*, pages 325–332, December 1988.

26. Seitz, Charles L. "The Cosmic Cube," *Communications of the ACM*, *28*(1):22–33 (1985).

27. Shannon, Robert E. *Systems Simulation: the Art and Science.* Prentice-Hall, 1975.

28. Stone, Harold S. *High Performance Computer Architecture.* Addison-Wesley Publishing Co., 1987.

29. Su, Wen-King. *Reactive-Process Programming and Distributed Discrete-Event Simulation.* Technical Report Caltech-CS-TR-89-11, California Institute of Technology, 1990.

30. Su, Wen-King and Charles L. Seitz. "Variants of the Chandy-Misra-Bryant Distributed Discrete-Event Simulation Algorithm," *Distributed Simulation 1989*, pages 38–43 (1989).

31. Wieland, Frederick and others. "Distributed Combat Simulation and Time Warp: The Model and its Performance," *Proceedings of the SCS Multiconference on Distributed Simulation*, pages 14–20 (1989).

Ann K. Lee was born in [redacted] She graduated in 1981 from Largo Senior High School in Largo, Maryland. She attended Howard University, Washington, DC, graduating in May 1985 with a B.S. in Mathematics, while obtaining her commission as a Second Lieutenant, U.S. Air Force, through the ROTC program. She was subsequently assigned to the 6545th Test Group, Air Force Systems Command, Hill AFB, Utah in the Improvement Engineering branch. In this assignment, she was the software engineer for system and software improvements of the Air Combat Manuevering Instrumentation system, an air-to-air combat training system located on the Utah Test and Training Range. She entered the Air Force Institute of Technology in June 1989.

Permanent address: [redacted]

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE December 1990	3. REPORT TYPE AND DATES COVERED Master's Thesis

4. TITLE AND SUBTITLE AN EMPIRICAL STUDY OF COMBINING COMMUNICATING PROCESSES IN A PARALLEL DISCRETE EVENT SIMULATION	5. FUNDING NUMBERS
6. AUTHOR(S) Ann Kathryn Lee, Capt, USAF	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Air Force Institute of Technology, WPAFB OH 45433-6583	8. PERFORMING ORGANIZATION REPORT NUMBER AFIT/GCS/ENG/90D-08
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Joint Tactical Fusion Program Management Office 1500 Planning Research Dr. McLean, VA 32102-5099	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

The primary goal of distributed discrete event simulations is to achieve speedup in simulation execution time by distributing the processing of the simulation over multiple processors. When partitioned for distribution in this fashion, simulations are typically partitioned such that there are more processes than processors. This thesis reviews existing methods for distributed discrete event simulations, and proposes general guidelines for efficient partitionings for a given communications topology based on empirical evidence. A performance analysis is conducted for two approaches to partitioning the system. The first method chosen is a mapping of multiple processes to a processor and the second approach utilizes a distributed event list approach, developed by Mannix. This approach combines smaller processes into a larger single process, incorporating a next event list similar to that used in a sequential simulation. Empirical studies compare the performance of the two approaches under a variety of conditions. The traditional Chandy-Misra approach to system partitioning is demonstrated to yield overall better performance than the distributed event list algorithm. General guidelines for partitioning the system for both approaches are developed based on the performance comparisons. The impact of system topology on simulation performance is demonstrated. A connection between the system topology and the communication protocol implementation is investigated. The distributed event list algorithm is shown to exhibit a performance anomaly for a system network with directed cycles, and a correction for the algorithm is proposed.

14. SUBJECT TERMS Simulation, Discrete Event Simulation, Distributed Simulation			15. NUMBER OF PAGES 87
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL